CAP 6615 - Neural Networks

Programming Assignment 1 -- Single-Layer Perceptron

Group member:

Bowei Wu (UFID: 75825722)

Haowen Chen (UFID: 81411485)

Tao Zhang (UFID: 76366624)

Xichen Li (UFID: 49287968)

Yangbo Yu (UFID: 29077422)

Yuwei Xia (UFID: 28267331)

Spring Semester 2022

28 Jan 2022

Table of Contents

1 Network Parameters

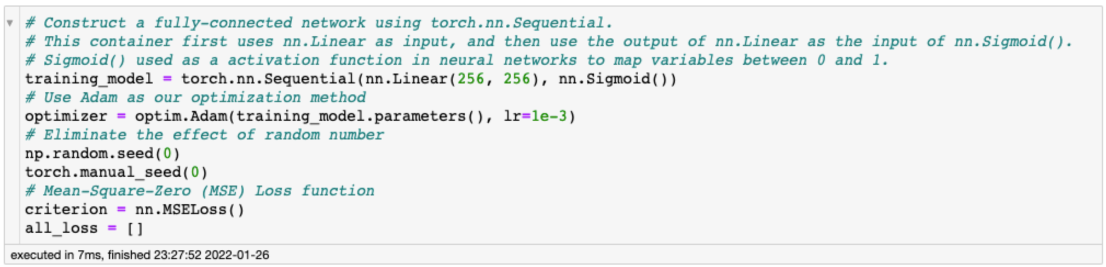
In this section, the parameters of our well-trained Single-Layer-Perceptron (SLP) will be specified.

- Inputs: A vector with size 256, containing 256 input values.
- Weights: 64K weights, in the form of a matrix with size [256, 256].
- Bias: Bias is a vector of size 256.
- Activation function: Sigmoid function.
- Optimizer: Adaptive Moment Estimation (Adam) is used as our optimization method, where the learning rate is 0.001.
- Output: Output is a vector including 256 values.

2 Python Code for SLP

In this part, we will display Python code generated for SLP. Before showing the code, there are a few details that we need to explain first.

We use a sequential container to construct this network. This container contains two modules, a fully connected layer as well as an activation function. Sigmoid function is chosen as activation function of our neural network because it exists between 0 and 1. By setting a threshold, for example, 0.5, we can replace all outputs smaller than 0.5 with 0 and for those equal to or larger than 0.5, we replace them with 1. In this way, we can regenerate output as an image with only black and white. Adam is used as optimization method and learning rate is 0.001. Figure 1is the SLP we designed and developed in Python using PyTorch.



```python
# Construct a fully-connected network using torch.nn.Sequential.
# This container first uses nn.Linear as input, and then use the output of nn.Linear as the input of nn.Sigmoid().
# Sigmoid() used as a activation function in neural networks to map variables between 0 and 1.
training_model = torch.nn.Sequential(nn.Linear(256, 256), nn.Sigmoid())
# Use Adam as our optimization method
optimizer = optim.Adam(training_model.parameters(), lr=1e-3)
# Eliminate the effect of random number
np.random.seed(0)
torch.manual_seed(0)
# Mean-Square-Zero (MSE) Loss function
criterion = nn.MSELoss()
all_loss = []
```

```
executed in 7ms, finished 23:27:52 2022-01-26
```

Figure 1: SLP constructed by PyTorch

After designing the SLP, we train and test it using 10 16*16-pixel images, where each image corresponds to a character from 0 to 9.

We first run some preliminary tests on the number of epochs that should be used for backpropagation procedure. We randomly pick 1, 10 and 100 as the number of epoch and the output results are shown in Figure 2, Figure 3, Figure 4 respectively. When epoch equals to 1, the output results looks like random generated image. This means that doing backpropagation only one time is far from enough to train this SLP. When epoch is increased to 10, some features are learned by SLP because the outputs show some regularity. When epoch is 100, the outputs are almost identical to our expected results.

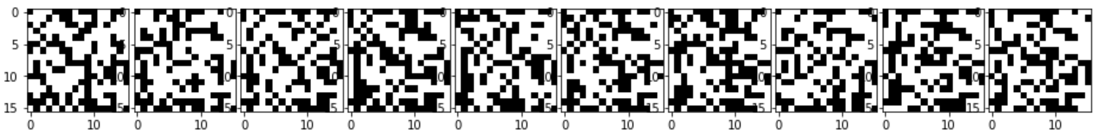


Figure 2: Testing results for epoch = 1

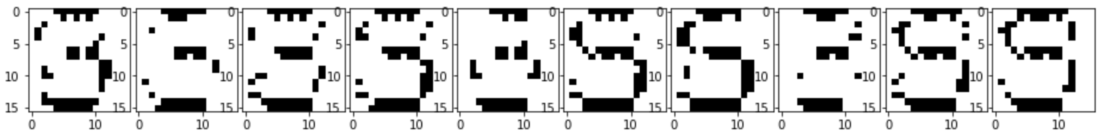


Figure 3: Testing results for epoch = 10

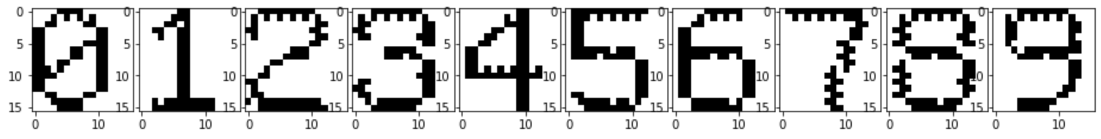


Figure 4: Testing results for epoch = 100

But by observing the tendency of loss function as shown in Figure 5, we think there might still be improvement by using a larger number of epoch. Hence, in our formal experiment, epoch is set as 500. Then we set step as 10, so we can print loss every ten epochs and see if our SLP is learning or not. The Python code for SLP training is in Figure 6.

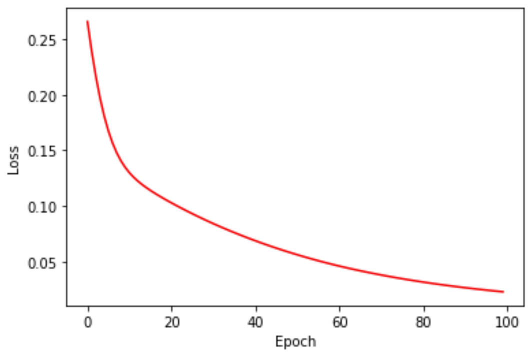
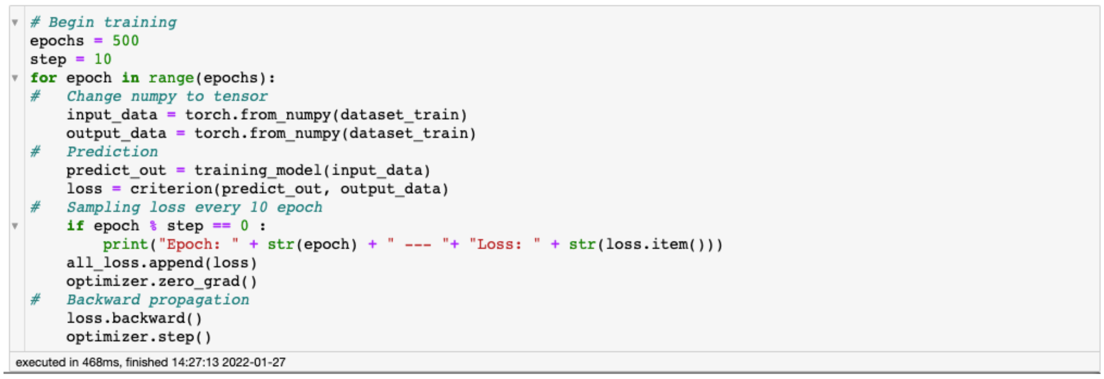


Figure 5: Loss of SLP training procedure when epoch = 100



```
# Begin training
epochs = 500
step = 10
for epoch in range(epochs):
#   Change numpy to tensor
    input_data = torch.from_numpy(dataset_train)
    output_data = torch.from_numpy(dataset_train)
#   Prediction
    predict_out = training_model(input_data)
    loss = criterion(predict_out, output_data)
#   Sampling loss every 10 epoch
    if epoch % step == 0 :
        print("Epoch: " + str(epoch) + " --- "+ "Loss: " + str(loss.item()))
    all_loss.append(loss)
    optimizer.zero_grad()
#   Backward propagation
    loss.backward()
    optimizer.step()
executed in 468ms, finished 14:27:13 2022-01-27
```

Figure 6: Python code for SLP training and epoch = 500

3 Training Set Configuration

In this section, the training set generated by us is displayed. Note that two rows of images are showed. The first row, Figure 7, is a screenshot of printed training images in jupyter notebook. The coordinates mean that the size of each image is 16*16-pixel. The images in the following row is the actual size of these 10 images, as shown in Figure 8.

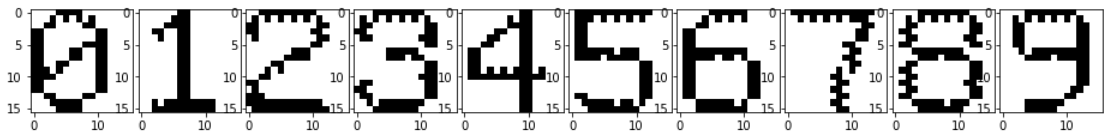


Figure 7: Screenshot of printed images in jupyter notebook

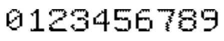


Figure 8: Actual images

4 SLP Output Results for Noiseless Inputs

Below are the output results for noiseless inputs. X-coordinate denotes value of Ffa and Y-coordinate is Fh. Note that only one point appears in this graph because all 10 outputs are overlapped with each other.

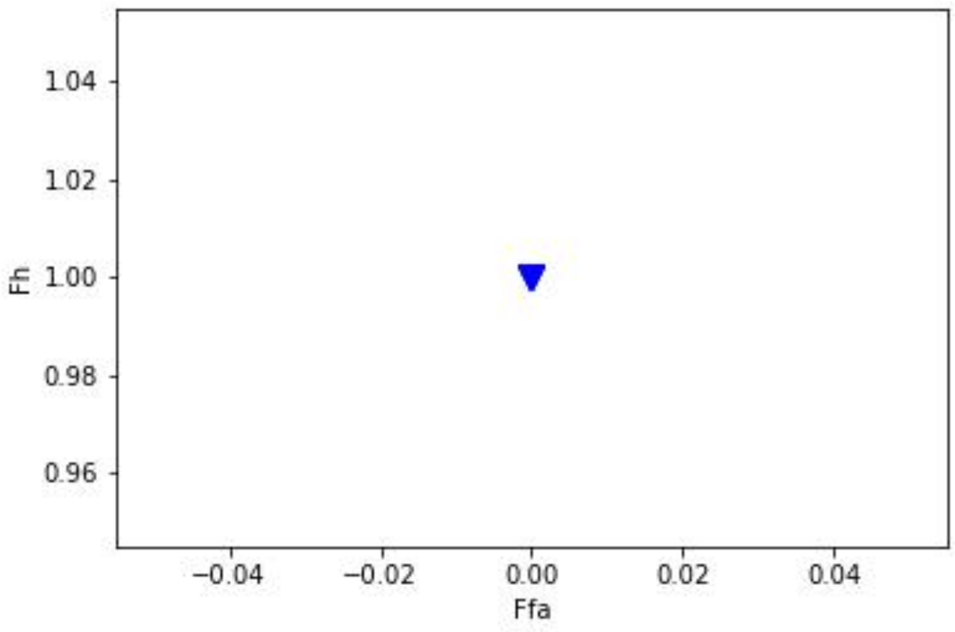


Figure 9: SLP output results for noiseless inputs

5 Pseudocode and Python Code for Fh and Ffa

In our project, we use 1 to represent white and 0 to represent black.

5.1 Pseudocode

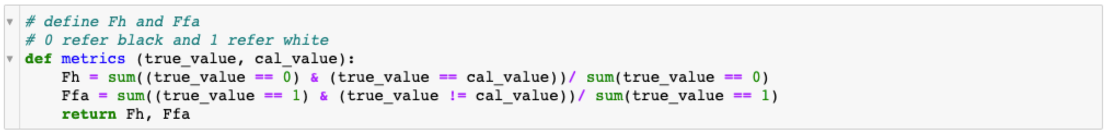
Suppose cal_values is a list containing all 10 test results and true_values is a list of 10 inputs.

Algorithm 1: Computing Fh and Ffa

```
Inputs: cal_values, true_values
Outputs: Fh, Ffa
1   $for\ i\ =\ 0\ to\ 9$:
2       $Fh[i] = \frac{number\ of\ 0\ in\ cal\_values[i]\ in\ correct\ place\ comparing\ with\ true\_values[i]}{total\ number\ of\ 0\ in\ true\_values[i]}$
3       $Ffa[i] = \frac{number\ of\ 0\ in\ cal\_values[i]\ in\ wrong\ place\ comparing\ with\ true\_values[i]}{total\ number\ of\ 1\ in\ true\_values[i]}$
```

5.2 Python code

We define a function called “metrics” to compute Fh and Ffa. As shown in Figure 10, this function has two input parameters, where true_value is input and cal_value is real test result.



```
# define Fh and Ffa
# 0 refer black and 1 refer white
def metrics (true_value, cal_value):
    Fh = sum((true_value == 0) & (true_value == cal_value))/ sum(true_value == 0)
    Ffa = sum((true_value == 1) & (true_value != cal_value))/ sum(true_value == 1)
    return Fh, Ffa
```

Figure 10: Python code for computing Fh and Ffa

6 SLP Output Results for Noise-corrupted Input

Figure 11 is the output results of Fh and Ffa for dataset with noise. This graph is a little different with the graph in section 4 because the X-coordinate represents Gaussian noise standard deviation from 0.0 to 0.10 and Y-coordinate stands for Fh and Ffa values. The black dots are Fh and white dots are Ffa.

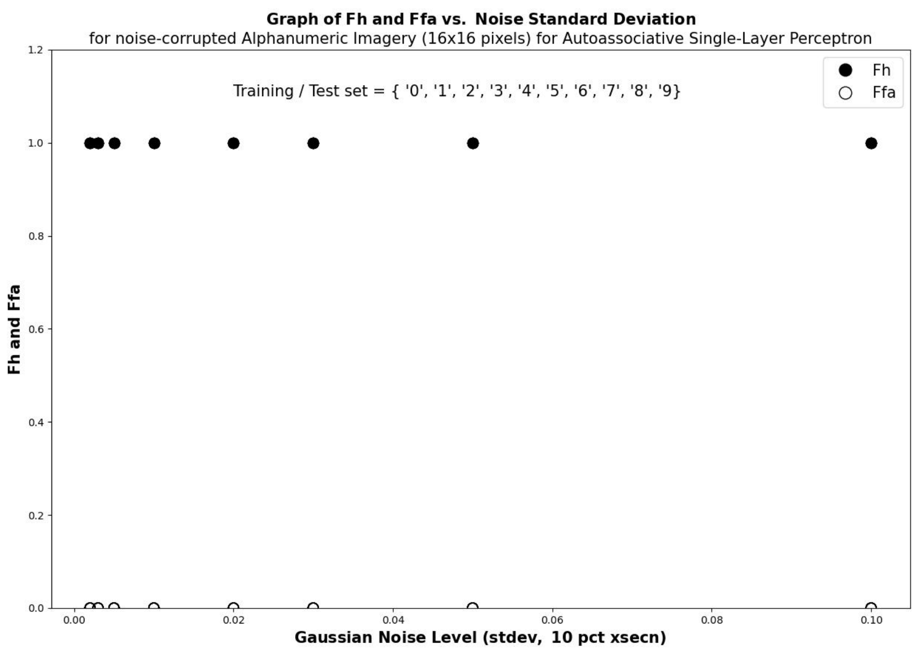


Figure 11: SLP output results for noise-corrupted input

7 Discussion and Improvements

From above description of our SLP, we suppose we can conclude that our SLP is successful in this assignment. First, it can function correctly and run as expected without producing any errors. Second, the test results are satisfying. Specifically, the ideal value of Fh is supposed to be 1 and Ffa 0. In both section 4 without noise and section 6 with noise, Fh and Ffa of our SLP all equal to ideal value.

But this success actually makes us think more. If the dataset is larger, can our SLP achieve the same performance? Since at present, the size of dataset is only 10, which is really small. If the size of dataset increases to 10,000 or even more, 100,000, what should we do to improve the performance of our network? Will adding more layers work? Will using other kinds of network work?

Despite of these thoughts, we also realize that there are more things we could do on this SLP. For instance, although we have run some preliminary tests to see how the number of epochs will affect the training result of our SLP, there are still lots of parameters that we can play with. We could try to use different activation functions or optimization methods and see what will happen.

Appendix

In this appendix, we design and train a fully connected Shallow Multilayer Neural Net (SMNN). Because some configurations including training set configuration and computation of Ffa and Fh are the same as SLP, we will omit those sections here.

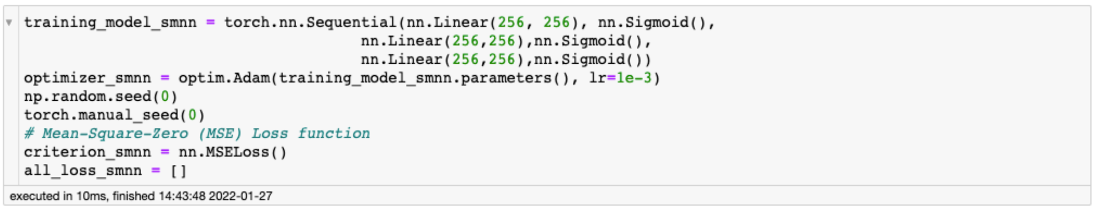
1 Network Parameter

In this section, we list the parameters of our three-layer SMNN.

- Inputs: A vector with size 256, containing 256 input values.
- Weights: 192K weights in total, and each layer has 64K weights in the form of a matrix with size [256, 256].
- Bias: Each layer has a bias vector of size 256.
- Activation function: Sigmoid function.
- Optimizer: Adaptive Moment Estimation (Adam) is used as our optimization method, where the learning rate is 0.001.
- Output: Output is a vector including 256 values.

2 Python Code for SMNN

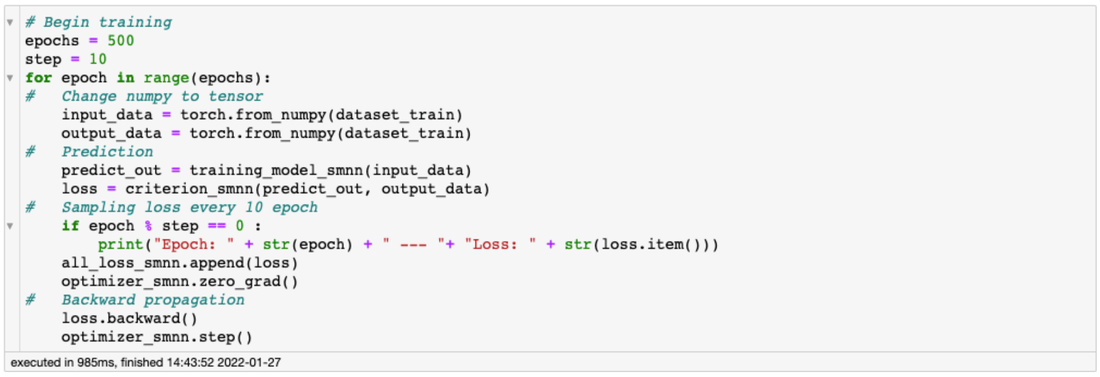
Figure 12 is Python code for generating SMNN. Figure 13 shows the Python code for training SMNN using original dataset before adding noise with epoch equals to 500. We also run some preliminary tests on the number of epochs and it turns out that when epoch is equivalent to 100, the performance of SMNN is worse than SLP as shown in Figure 14.



```
training_model_smnn = torch.nn.Sequential(nn.Linear(256, 256), nn.Sigmoid(),
                                          nn.Linear(256,256),nn.Sigmoid(),
                                          nn.Linear(256,256),nn.Sigmoid())
optimizer_smnn = optim.Adam(training_model_smnn.parameters(), lr=1e-3)
np.random.seed(0)
torch.manual_seed(0)
# Mean-Square-Zero (MSE) Loss function
criterion_smnn = nn.MSELoss()
all_loss_smnn = []
```

executed in 10ms, finished 14:43:48 2022-01-27

Figure 12: Python code for SMNN



```
# Begin training
epochs = 500
step = 10
for epoch in range(epochs):
#   Change numpy to tensor
    input_data = torch.from_numpy(dataset_train)
    output_data = torch.from_numpy(dataset_train)
#   Prediction
    predict_out = training_model_smnn(input_data)
    loss = criterion_smnn(predict_out, output_data)
#   Sampling loss every 10 epoch
    if epoch % step == 0 :
        print("Epoch: " + str(epoch) + " --- "+ "Loss: " + str(loss.item()))
    all_loss_smnn.append(loss)
    optimizer_smnn.zero_grad()
#   Backward propagation
    loss.backward()
    optimizer_smnn.step()
executed in 985ms, finished 14:43:52 2022-01-27
```

Figure 13: Python code for SMNN training when epoch = 500

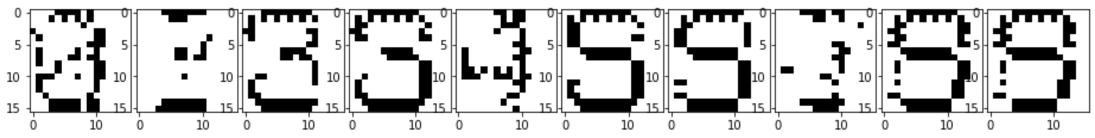


Figure 14: Testing results of SMNN when epoch = 100

3 SMNN Output Results for Noiseless Inputs

Figure 15 is the output results for noiseless inputs using SMNN. Note that only two points appear in this figure because there are 8 points overlapping at (0.00, 1.000).

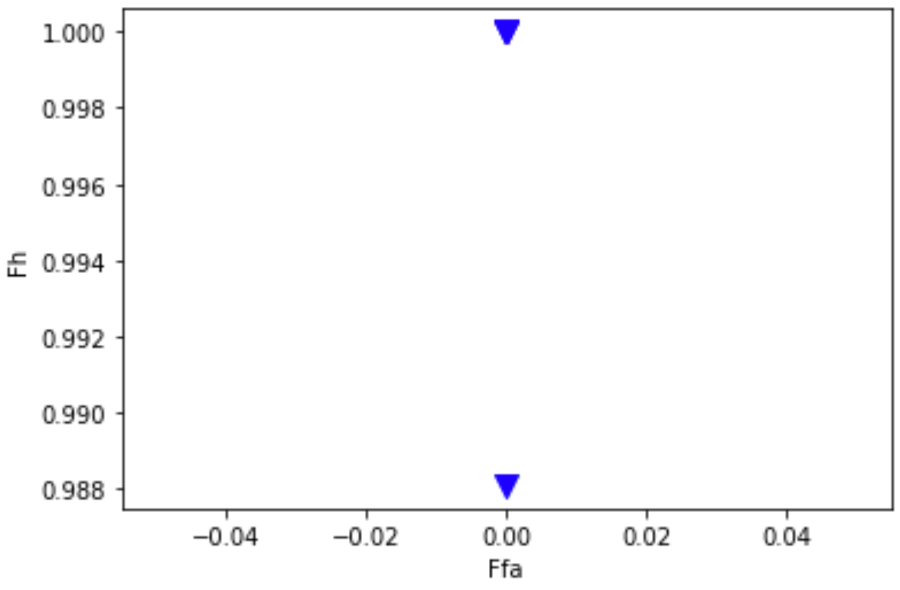


Figure 15: SMNN output results for noiseless inputs

4 SMNN Output Results for Noise-corrupted Input

Figure 16 is output results for dataset with noise using SMNN.

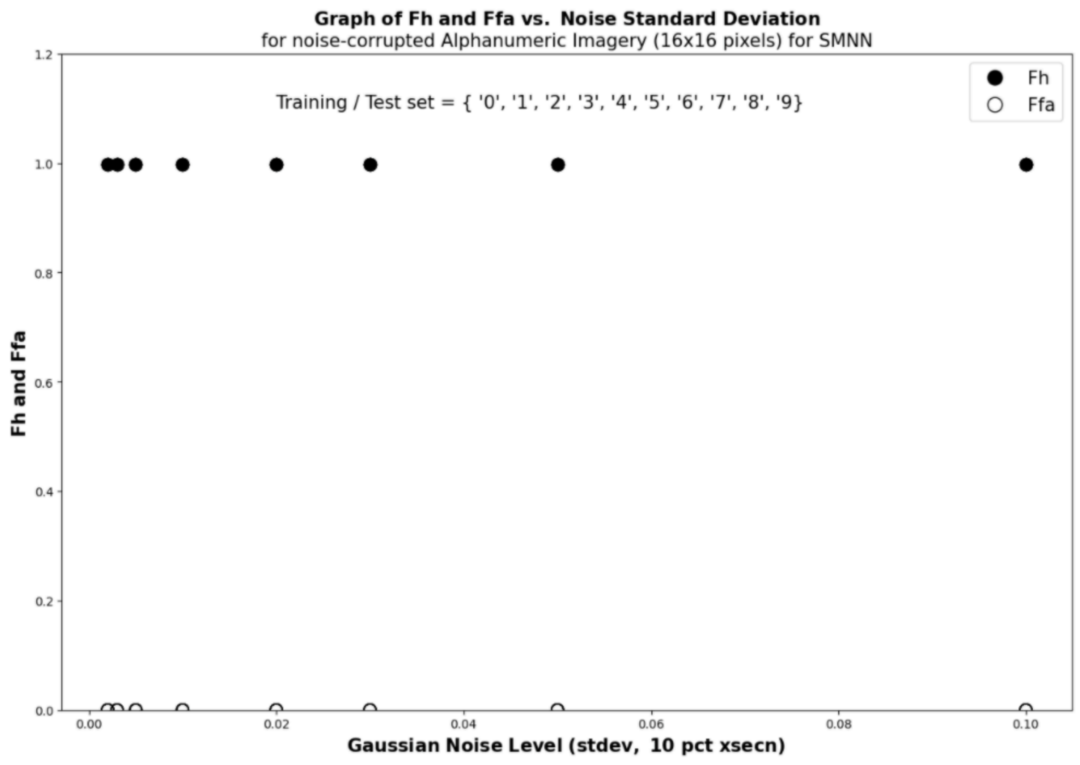


Figure 16: SMNN output results for noise-corrupted input

5 Discussion and Improvements

In this appendix, we build a three-layer fully connected neural network. Actually, this SMNN does not have much difference from the SLP, except it has two more layers.

Regarding the performance of SLP and SMNN, we hoped the performance of SMNN should be greater than SLP after training the same number of epochs. However, if set epoch as 100, we surprisingly to find that SLP actually performs better in preliminary tests than SMNN, as shown in Figure 4 and Figure 14. But when epoch equals to 500, both SLP and SMNN can do well in testing. To improve this SMNN, we might try to use different activation functions and different optimization methods in future work.