CAP 6615 - Neural Networks - Programming Assignment 2 – Multi-Layer Neural Network

Zhounan Li, Huaiyue Peng, Tongjia Guo, Zhiyun Ling, Mingjun Yu

Spring Semester 2021
26 Feb 2021

1 Network parameters

1.1 About SIFT

Although SIFT is a great algorithm to extract features from images, in this project we choose not to use SIFT. There are two reasons. The first one is that the image is too small (16 * 16) to do SIFT. After resizing the images to larger images (300 * 300), there are many jagged shapes on the edges of the characters. And unfortunately, SIFT will focus more on these shapes.

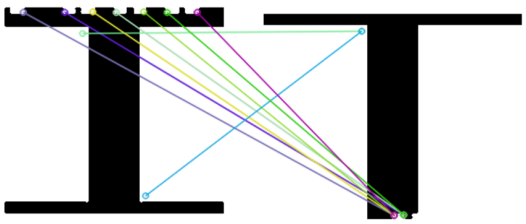

Figure 1: The results of SIFT, this is not a good phenomenon for us to extract the features

If we don't care about the phenomenon that mentioned above, we use the features to compute a similarity matrix using cosine similarity, then the matrix will look like this:

Even in a pair that we intuitively believe that the similarity should be high, the similarity is also very low. For the reasons above, we choose not to use SIFT is our project.

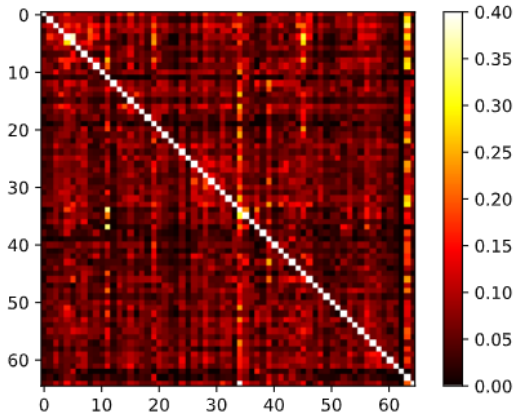

Figure 2: The results of similarity matrix, this is not a good phenomenon for us to extract the features

1.2 Parameters of Simple Network with Scalar Output

In this method, we use 32 images as our train dataset. The input size is 16*16 = 256. We have two hidden layers. The size of first hidden layer is 512 and the size of second hidden layer is 128. We use Relu function as our activation function because Relu is a linear function, which can learn relationships in data in deep neural network. Relu function is

$$f(x) = max(0, x) \tag{1}$$

After second hidden layer, we use Sigmod function to project the results in interval [0,1]. The output is a 65-size one-hot vector. In this vector, only the index of train images we choose has values. Other index are zero or equal to zero. The index of the output pattern is found by locating the maximum value in output one-hot vector. For example, if the maximum element is at index 1. Then we think this image belongs to class 1. The loss function we use is categorical crossentropy because there are more than two classes and we expect to get one-hot vector in out output.

1.3 Fully-connected Network with Image Output

In this method, our input size is 16*16 = 256 so there are 64k weights. We have two hidden layers. Both of these two layers are 512-size and we use Relu function as our activation function. The output size is 256, which is the same as input images. We use sigmod function to make 256 output nodes vary over the interval [0,1]. The loss function we use is MSE because we need to compare the output image and input image and find how different they are. The MSE function is

$$MSE = \frac{1}{n}\sum_{i=1}^{i=n}(y_i - \hat{y}_i)^2 \tag{2}$$

2 Python code for MLNNs

2.1 Python code for Parameters of Simple Network with Scalar Output

```
def get_model_approach_1():
    img_input = tf.keras.Input(shape=(256, ))
    hidden1 = Dense(512, activation='relu')(img_input)
    hidden2 = Dense(128, activation='relu')(hidden1)
    output = Dense(65, activation='softmax')(hidden2)
    model = tf.keras.Model(img_input, output)
    model.compile(optimizer='sgd',)
    loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

2.2 Python code for Fully-connected Network with Image Output

```
def get_model_approach_2():
    img_input = tf.keras.Input(shape=(256, ))
    hidden1 = Dense(512, activation='relu')(img_input)
    hidden2 = Dense(512, activation='relu')(hidden1)
    output = Dense(256, activation='sigmoid')(hidden2)
    model = tf.keras.Model(img_input, output)
    # model.compile(optimizer='adam',)
    loss='binary_crossentropy', metrics=['accuracy'])
    model.compile(optimizer='adam', loss='MSE')
    return model
```

3 Training set configuration

We choose 32 images A, B, C, D, E, H, I, K, L, M, N, O, P, Q, R, S, U, V, W, X, Y, Z, b, d, e, g, h, i, l, t, 4 and : to be our train dataset. We randomly choose 16 images as validation dataset. We choose these images because they are similar to the remaining images, for example, I and l is similar to 1 and D and O is similar to 0. So we choose these images so that when we face different images in test process, we can still get good performance. The remaining images, except for ", are all in test dataset. Then we resize these pictures in 16*16 pixels images.

Our code for building dataset in method one:

Figure 3: The images in our dataset

```
def get_dataset1(train_idx, test_idx, add_noise):
    train_set = [images_small[i] for i in train_idx]
    test_set = [images_small[i] for i in test_idx]
    train_label = np.array([[0] * 65 for i in range(32)])
    test_label = np.array([[0] * 65 for i in range(32)])
    train_label.shape
    for i in range(32):
        train_set[i] = np.reshape(train_set[i] / 255, (256,
        )).astype('int').astype('float64')
        train_label[i][train_idx[i]] = 1
    train_set = np.array(train_set)

    for i in range(32):
        test_set[i] = np.reshape(test_set[i] / 255, (256,
        )).astype('int').astype('float64')
        test_label[i][test_idx[i]] = 1
    test_set = np.array(test_set)
```

Our code for building dataset in method two:

```
def get_dataset2(train_idx, test_idx, add_noise):
    train_set = [images_small[i] for i in train_idx]
    test_set = [images_small[i] for i in test_idx]
    train_label = [images_small[i] for i in train_idx]
    test_label = [images_small[i] for i in test_idx]

    for i in range(32):
        train_set[i] = np.reshape(train_set[i] / 255, (256,
        )).astype('int').astype('float64')
        train_label[i] = np.reshape(train_label[i] / 255,
        (256, )).astype('int')
    train_set = np.array(train_set)
    train_label = np.array(train_label)

    for i in range(32):
```

```
        test_set[i] = np.reshape(test_set[i] / 255, (256,
        )).astype('int').astype('float64')
        test_label[i] = np.reshape(test_label[i] / 255, (256,
        )).astype('int')
    test_set = np.array(test_set)
    test_label = np.array(test_label)
```

4 MLNNs output results for noiseless input

4.1 Results of Simple Network with Scalar Output

After building our own dataset, we use our model to train the images we choose in last section. After training we use the following equations to calculate Fh and Ffa.

$$Fh = \frac{number\ of\ black\ pixels\ in\ output\ image\ that\ occur\ in\ correct\ places}{total\ number\ of\ black\ pixels\ in\ input\ image} \tag{3}$$

$$Fh = \frac{number\ of\ black\ pixels\ in\ output\ image\ that\ occur\ in\ wrong\ places}{total\ number\ of\ white\ pixels\ in\ input\ image} \tag{4}$$

After training, we get our loss function and the accuracy of our model. In this section, we set epoches to 100 because the loss function reaches convergence when epoch is 100. The accuracy reaches 1 when epoch reaches 100, which means our model has very high accuracy on the train dataset.

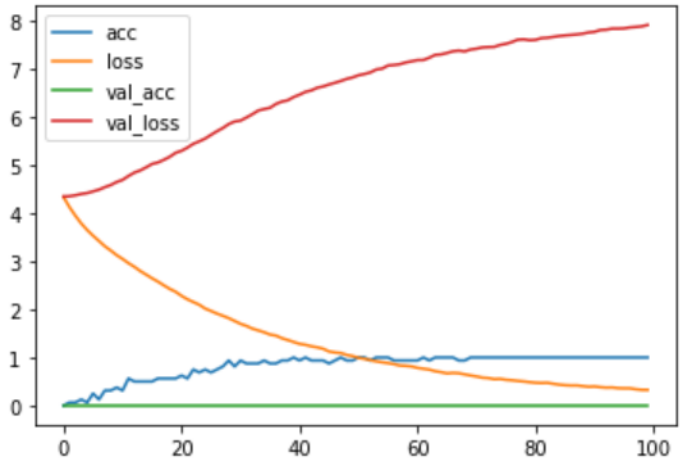

Figure 4: The results of noiseless data in method one

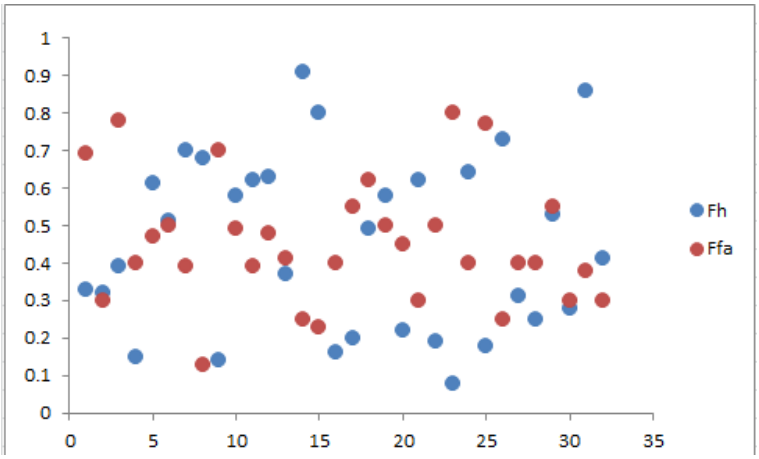

Figure 5: The Fh and Ffa of noiseless data

The validation loss is rising as epoch increases. This is because we make more mistakes on validation set. And our model will not get best performance on train set. Our model will be simpler and this will also avoid over-fitting.

4.2 Results of Fully-connected Network with Image Output

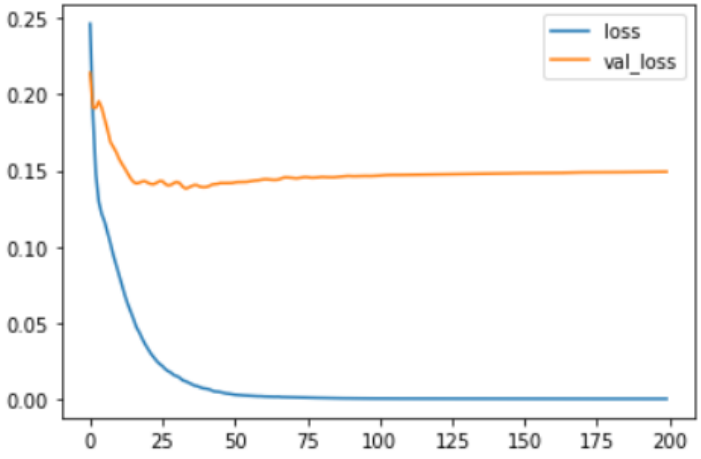

Figure 6: The results of noiseless data in method two

From results, we can see that our validation loss remains 0.15, not 0. This means our model still make some mistakes on validation set and will avoid over-fitting.

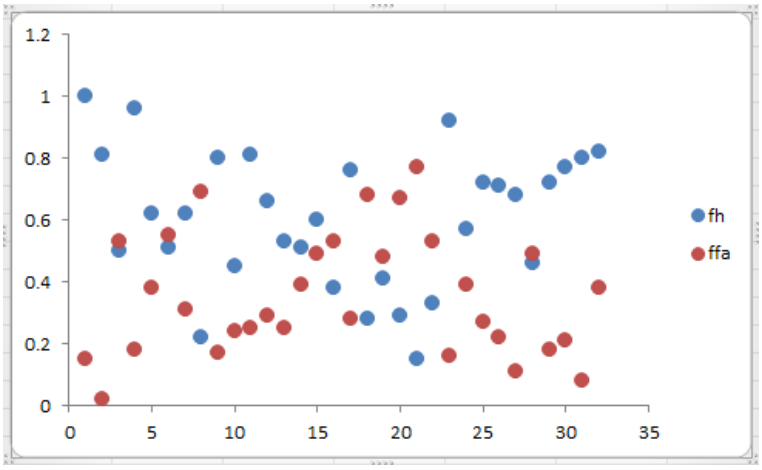

Figure 7: The Fh and Ffa of noiseless data

5 Code for Computing Fh and Ffa

```
# get Fh and Fha
    Fh = {}
    Fha = {}

    for i in range(64):
        for j in range(64):
            arr1 = np.reshape(images_small[i] / 255, (256,
            )).astype('int')
            arr2 = np.reshape(images_small[j] / 255, (256,
            )).astype('int')
            hit = np.sum((1-arr1) & (1-arr2))
            total = np.sum(1-arr1)
            pred = np.sum(1-arr2)
            Fh[str(i)+str(j)] = hit / total
            Fha[str(i)+str(j)] = (pred-hit) / pred

#the code for graph Fh and Ffa
matrix = []
dev = [0, 0.001, 0.002, 0.003, 0.005,  0.01, 0.02, 0.03, 0.05,
0.1]
for d in dev:
    fhs, fhas = train_process1(train_idx, test_idx, d)
    matrix.append(fhs)
    matrix.append(fhas)
matrix = np.transpose(np.array(matrix))
print(matrix)
samples = [label2name[i] for i in test_idx]
```

```
header = []
for n in dev:
    header.append((str(n), 'Fh'))
    header.append((str(n), 'Fha'))
print(header)
header = pd.MultiIndex.from_tuples(header)
df = pd.DataFrame(matrix, dtype='float', index=samples,
columns=header)
df = df.round(2)
df.to_excel("metric_1.xlsx", index = True, header=True)

print(matrix)
print(samples)

# %%
cols = []
for n in dev:
    cols.append((str(n), 'Fh'))
test_Fh = np.transpose(df[cols].values)
cols = []
for n in dev:
    cols.append((str(n), 'Fha'))
test_Fha = np.transpose(df[cols].values)

# %%
plt.figure(figsize=(10, 10))

for i in range(10):
    fh = test_Fh[i]
    fha = test_Fha[i]
    plt.scatter([i]*32, fh, color='r')
    plt.scatter([i]*32, fha, color='b')

plt.xticks(np.arange(10), dev)
plt.xlabel('dev of noise')
plt.show()
```

6 Code for Computing ROC

```
#Compute ROC
pred = model.predict(test_set)
plt.figure(figsize=(16, 16))
for i in range(32):
```

```
    y_pred = pred[i]
    y_true = test_label[i]
    fpr, tpr, threshold = metrics.roc_curve(y_true, y_pred)
    roc_auc = metrics.auc(fpr, tpr)
    plt.title('ROC')
    plt.plot(fpr, tpr, label = label2name[test_idx[i]]+' AUC
    = %0.2f' % roc_auc)
    plt.legend(loc = 'lower right')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
plt.show()
```

7 Output results for noise-corrupted input

7.1 Results of Simple Network with Scalar Output

After training with noiseless data, we add noise to the input data. Noise is Gaussian-distributed with 10 percent cross-section(i.e. There are 25 noise pixels in this project because there are 256 pixels in input data.) and have zero mean, and standard deviation of 0.001, 0.002, 0.003, 0.005, 0.01, 0.02, 0.03, 0.05, and 0.1.

7.2 Results of Fully-connected Network with Image Output

From the above results, we can see that our model performs well. In most images, the Fh is bigger than 0.5 and Ffa is smaller than 0.5.

8 Discussion

From the above results, we know that our model performs very well on train and test dataset. But we still have many aspects to improve. First, the direct use of SIFT algorithm doesn't perform well in this project because the image is very small. We need to resize these images. But there are many jagged shapes on the edges of the characters and will make algorithm not perform well. So we also need to do some correction, such as changing some white pixel to black pixle, to resized images. In this project, we have validation dataset to avoid over-fitting. But when test dataset becomes bigger. A validation dataset may not avoid over-fitting. So a regularization may be needed. Our model uses two hidden layers. Though it performs well in this project. But if data becomes more complicated. Our network may be too shallow to do classification. So make our network deeper is our main aim in the future. But deeper network also means more time to train and test.

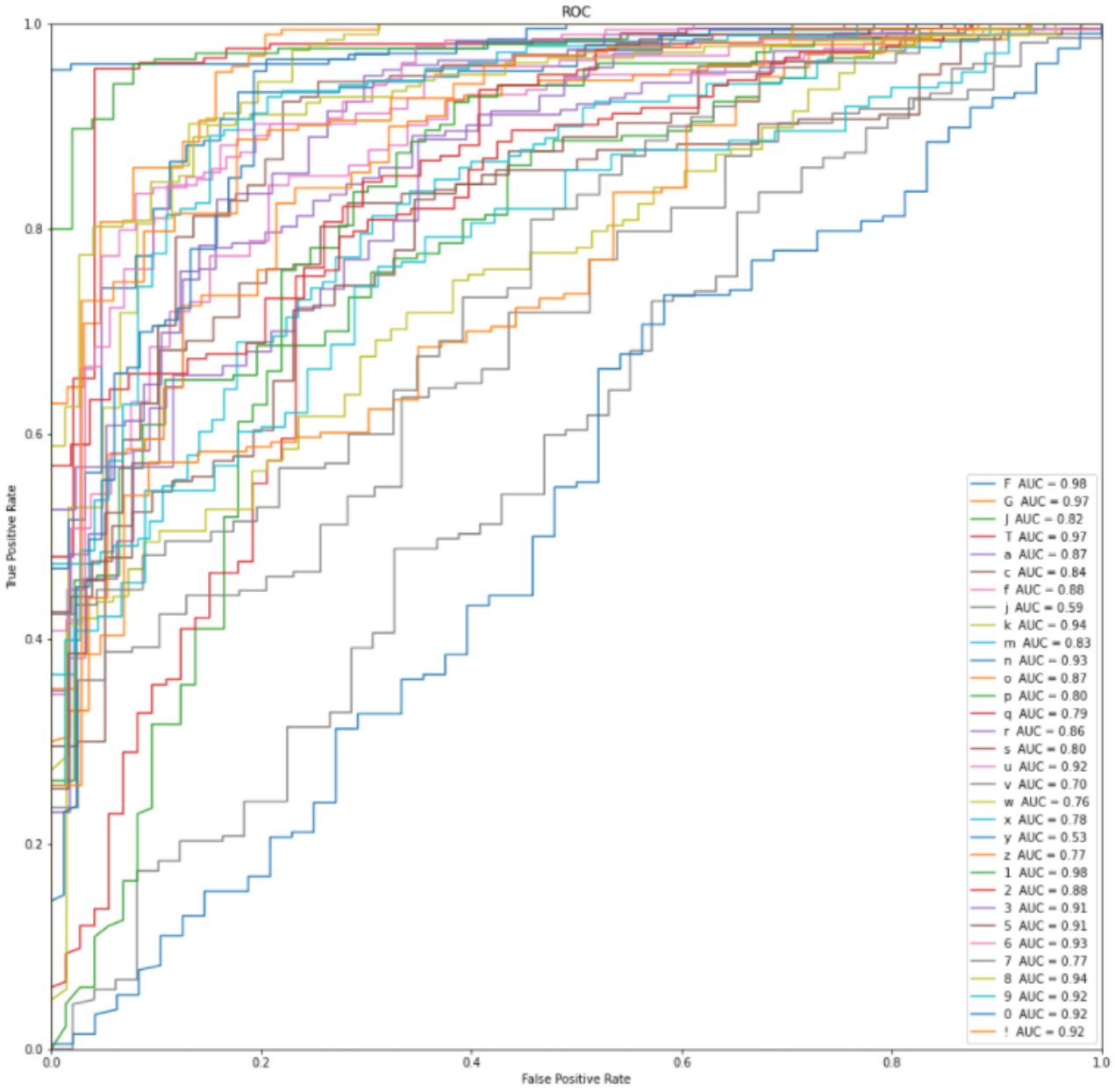

Figure 8: The ROC of noiseless data

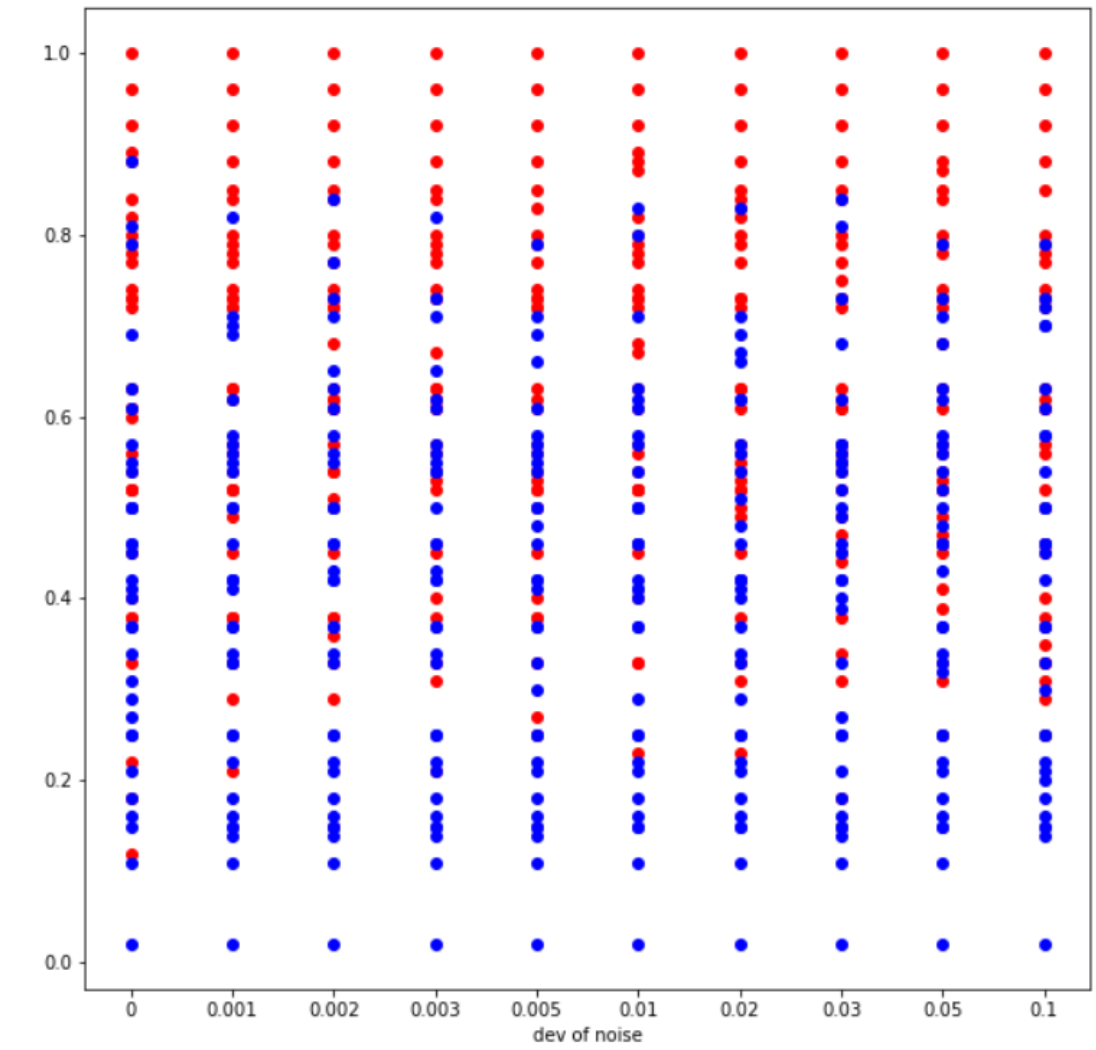

Figure 9: The results of noise data with different std in graph

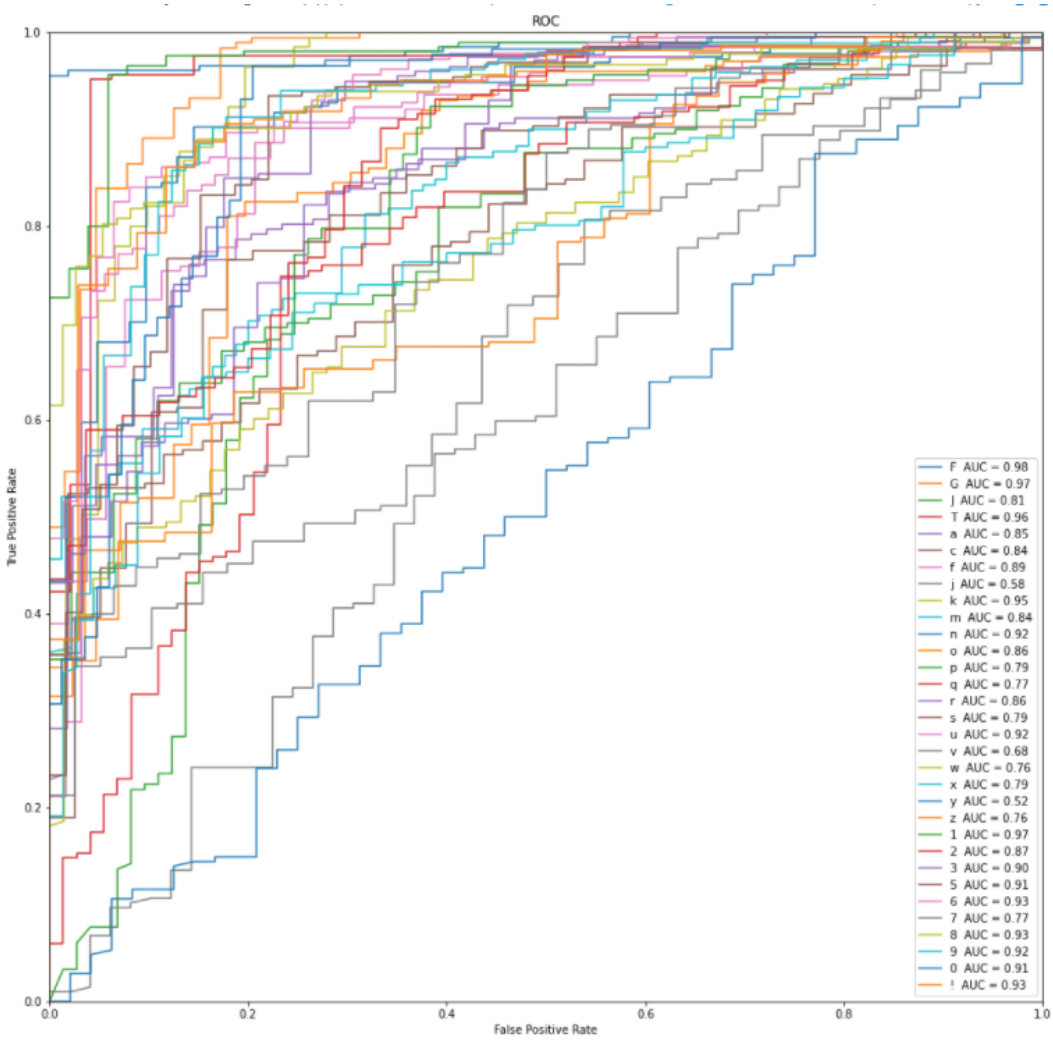

Figure 10: The results of ROC with std = 0.001

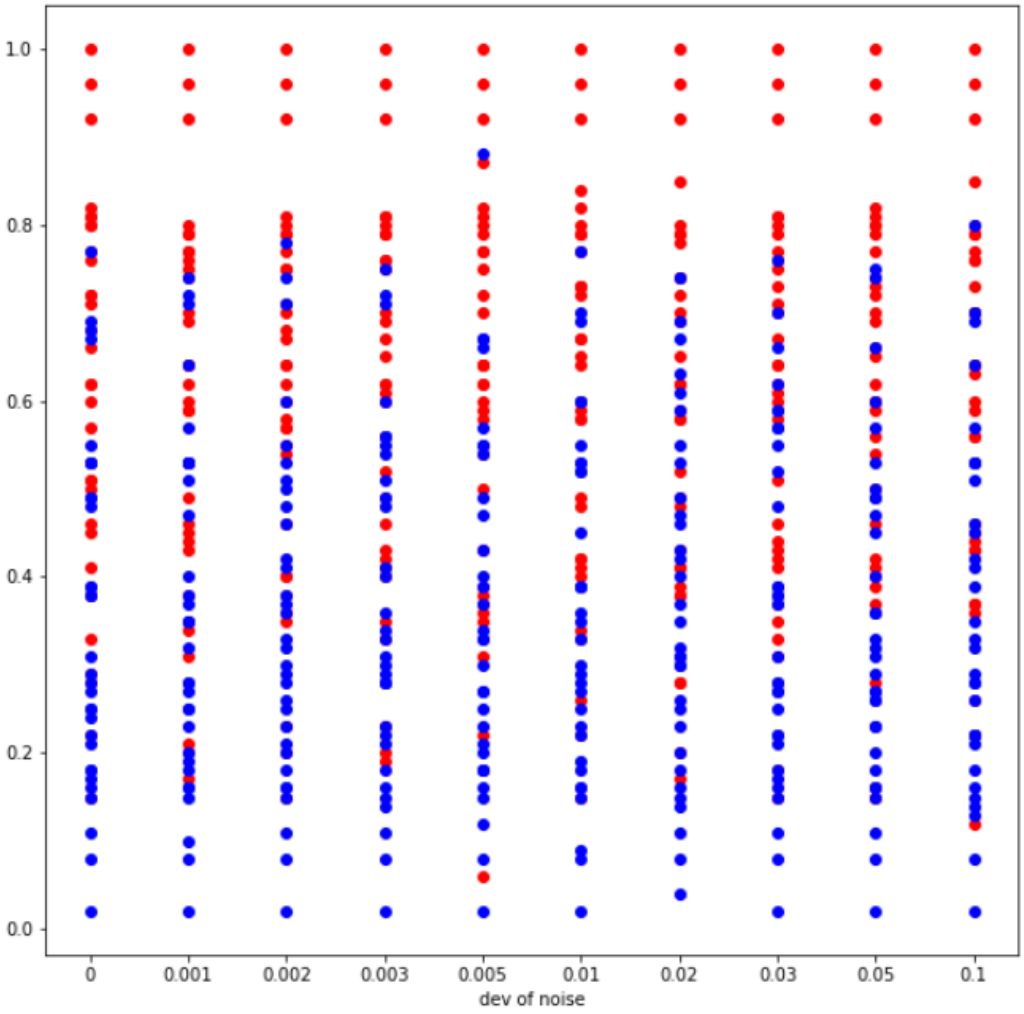

Figure 11: The results of noise data with different std in graph