Project Report

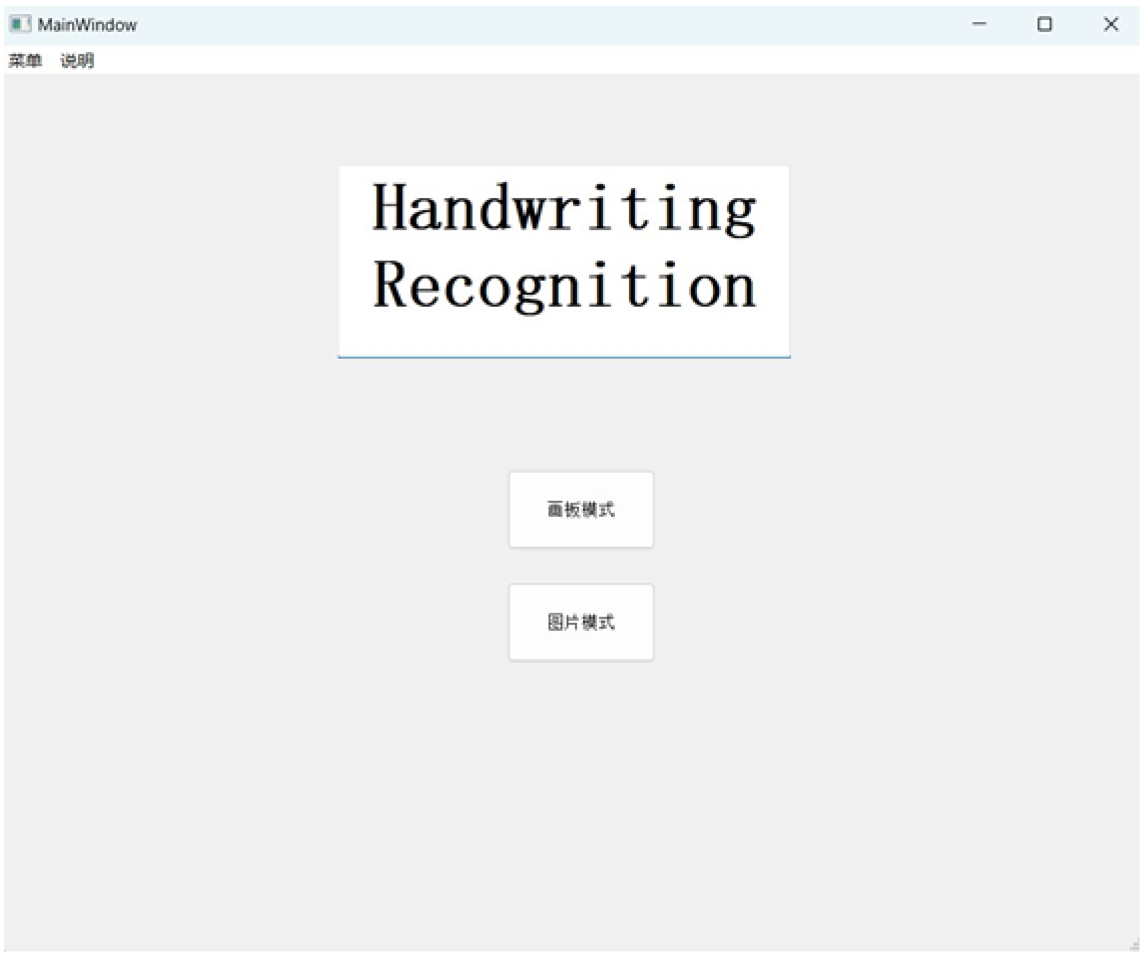
1.Introduction

The Handwriting Numerals Recognition is the most intriguing example for our group during the whole season learning process. Therefore, we decide to reproduce this example and want to try something more difficult ---- Handwriting Letter Recognition.

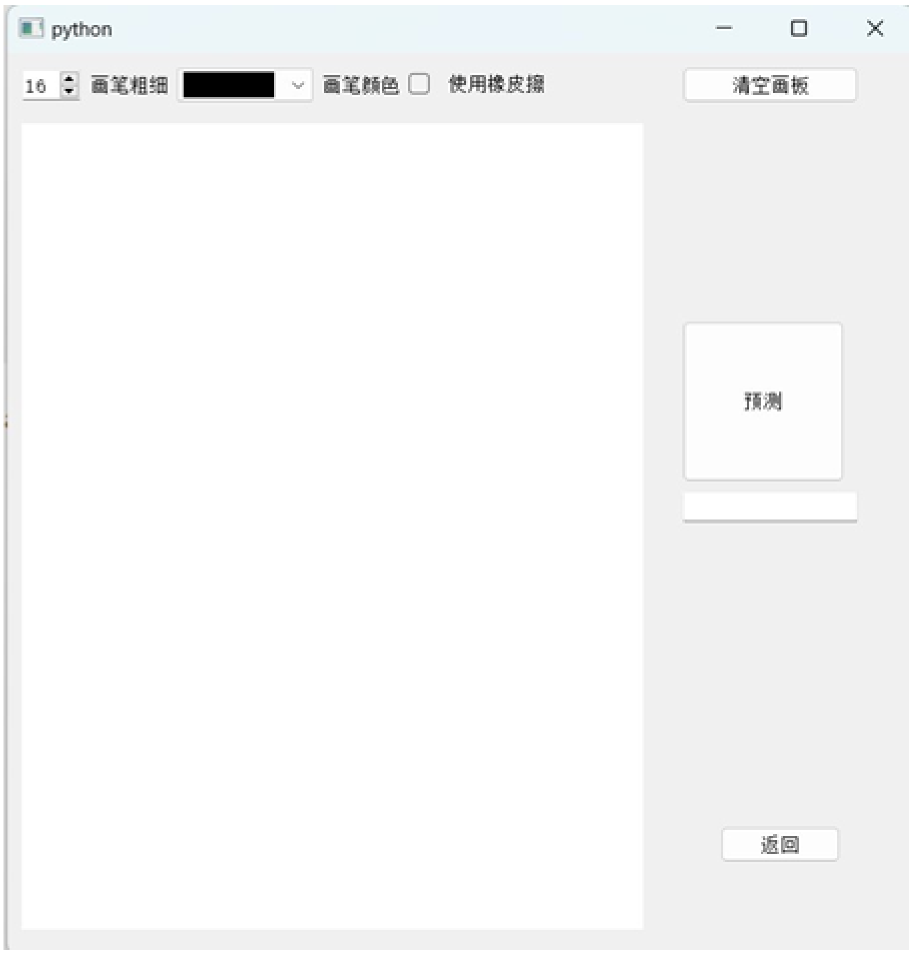
We not only use KNN to solve this problem, but also use LeNet and AlexNet which are the basic models in CNN and we think this is a good way to help us learn CNN and get used to it.

The first job we have done is comparison. We have known that KNN can be implemented in handwriting numerals recognition, but how is the effect of KNN in handwriting letter recognition with much more labels? So we compare the accuracy of the three models of the same test set after training with the same training set.

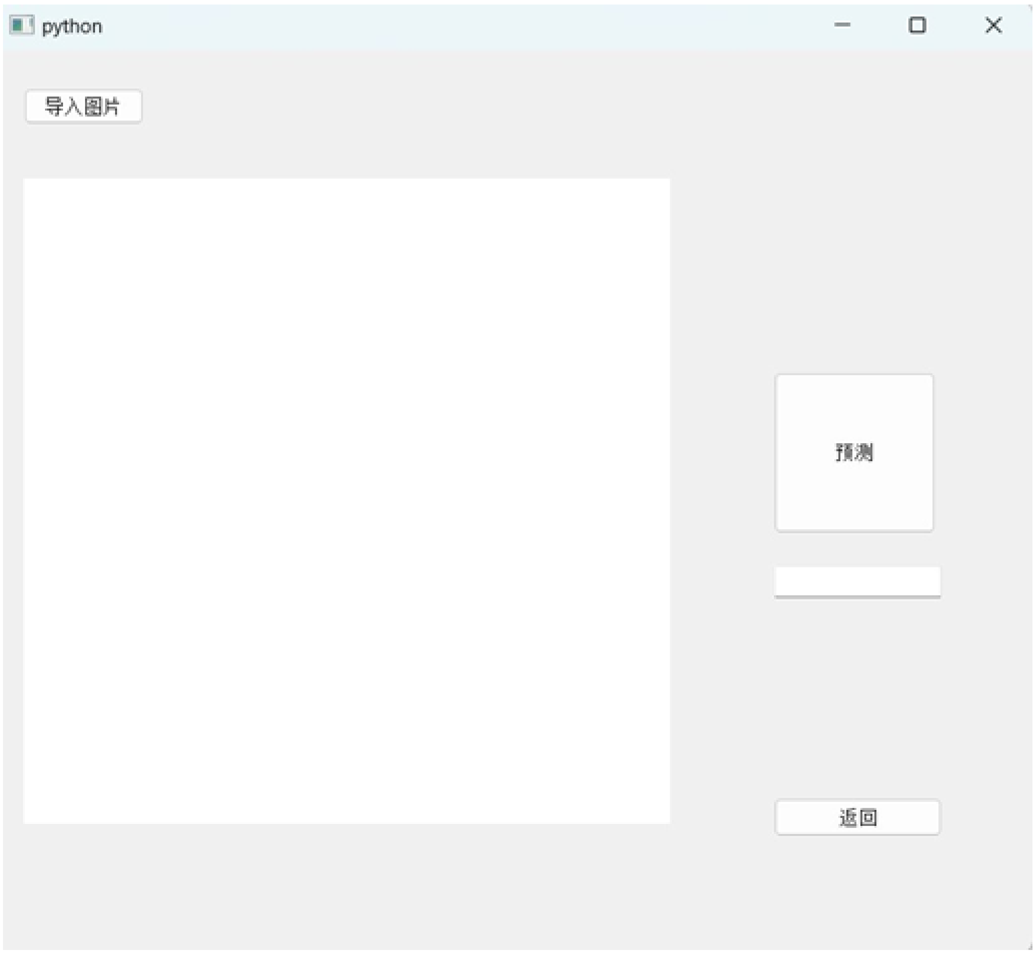
And the second job is that we write a simple python program to make the recognition easy to operate. We have two mode in our program: Picture Mode and Paint Board Mode.



In the Paint Board Mode, we can use our own brush to draw a letter and transmit to the AlexNet to recognize.



And in the Picture Mode, we can import a letter image from our computer and use AlexNet to recognize.

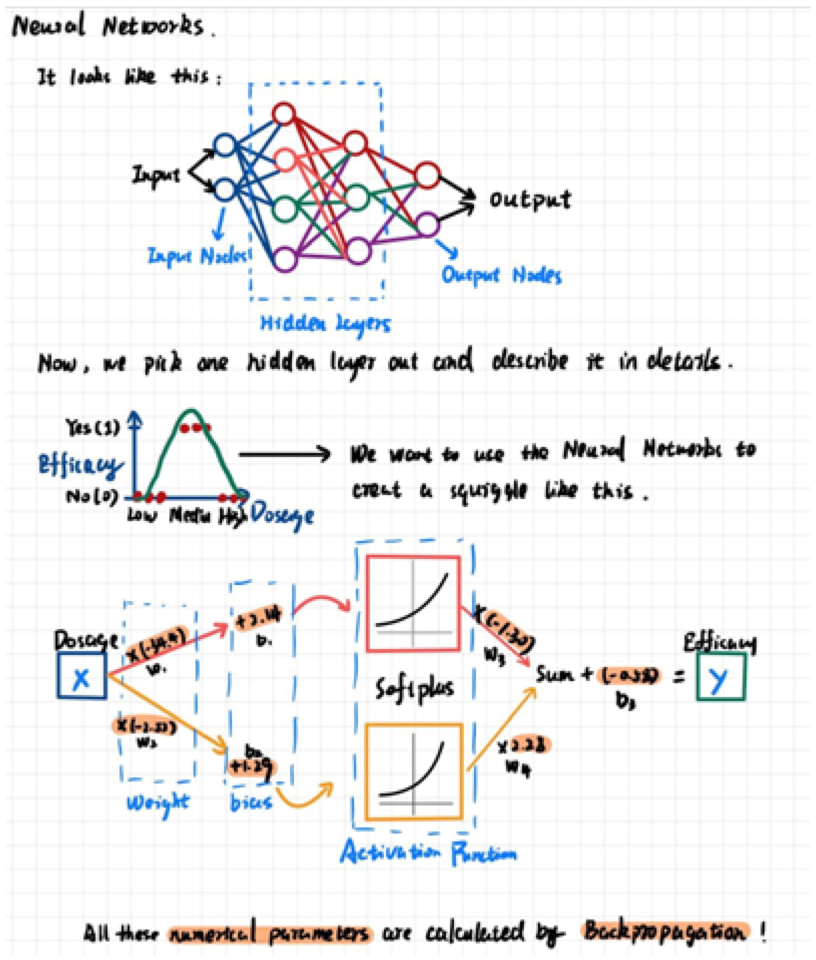


2 CNN

This part is mainly about how we learn CNN and what knowledge points have we learned.

To begin with, we need to know **What is the neural network?**

2.1 Neural Network

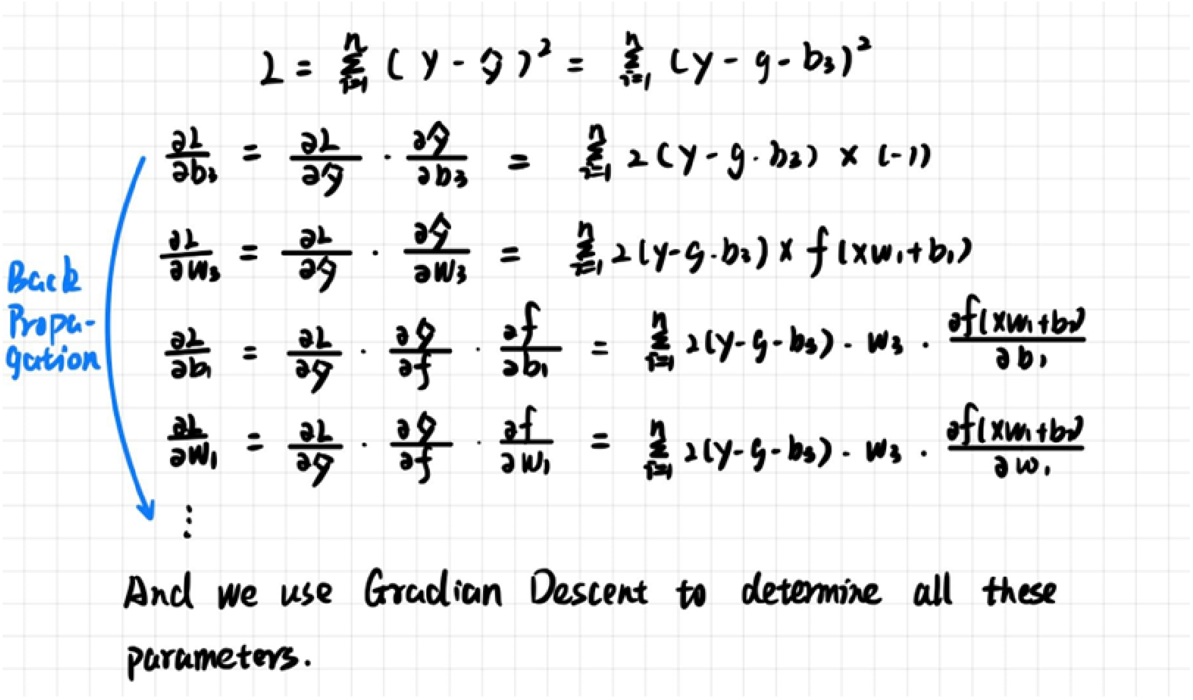

Neural Networks.

It looks like this:

Now, we pick one hidden layer out and describe it in details.

We want to use the Neural Networks to creat a squiggle like this.

All these numerical parameters are calculated by Backpropagation!



$$L = \sum_{i=1}^{n} (y - \hat{y})^2 = \sum_{i=1}^{n} (y - g - b_3)^2$$

Back Propa-gation

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b_3} = \sum_{i=1}^{n} 2(y - g - b_3) \times (-1)$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_3} = \sum_{i=1}^{n} 2(y - g - b_3) \times f(xw_1 + b_1)$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial b_1} = \sum_{i=1}^{n} 2(y - g - b_3) \cdot w_3 \cdot \frac{\partial f(xw_1 + b_1)}{\partial b_1}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f} \cdot \frac{\partial f}{\partial w_1} = \sum_{i=1}^{n} 2(y - g - b_3) \cdot w_3 \cdot \frac{\partial f(xw_1 + b_1)}{\partial w_1}$$

$$\vdots$$

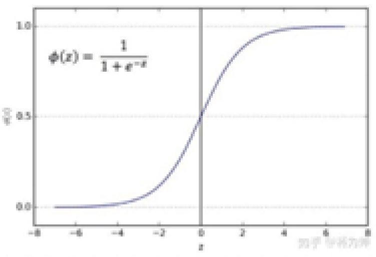
And we use Gradian Descent to determine all these parameters.

The basic structure is as above. And next part we will introduce some usual activation functions.

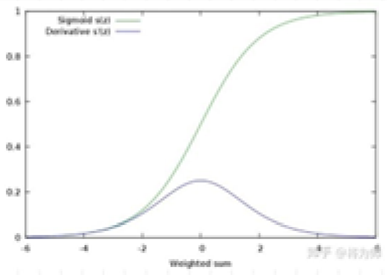
2.2 Activation Function

1. Sigmoid and Vanishing Gradient

The graph for a sigmoid function looks like this:



从图像可以看到：函数两个边缘的梯度约为0，梯度的取值范围为(0, 0.25).



当我们用BP算法计算每一个权重时，我们会发现sigmoid函数的梯度是其中一个因子.

$$\frac{\partial(\sigma(w^Tx+b))}{\partial w} = \boxed{\frac{\partial(\sigma(w^Tx+b))}{\partial(w^Tx+b)}} \cdot \frac{\partial(w^Tx+b)}{\partial w}$$

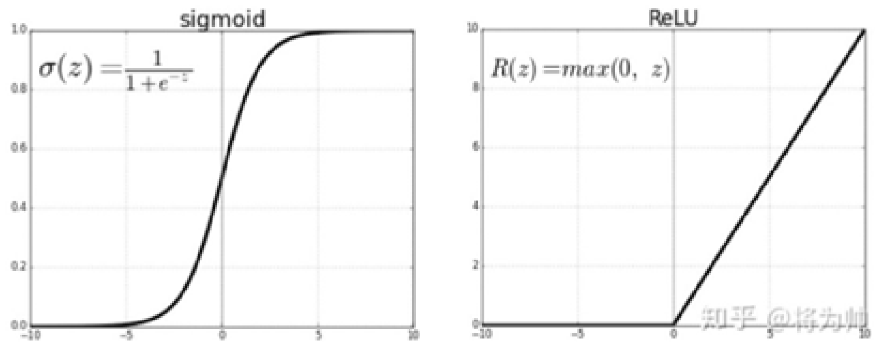
$\rightarrow \sigma\ (0, 0.25)$

2. ReLU (Rectified Linear Unit)

修正线性单元 / 整流线性单元函数

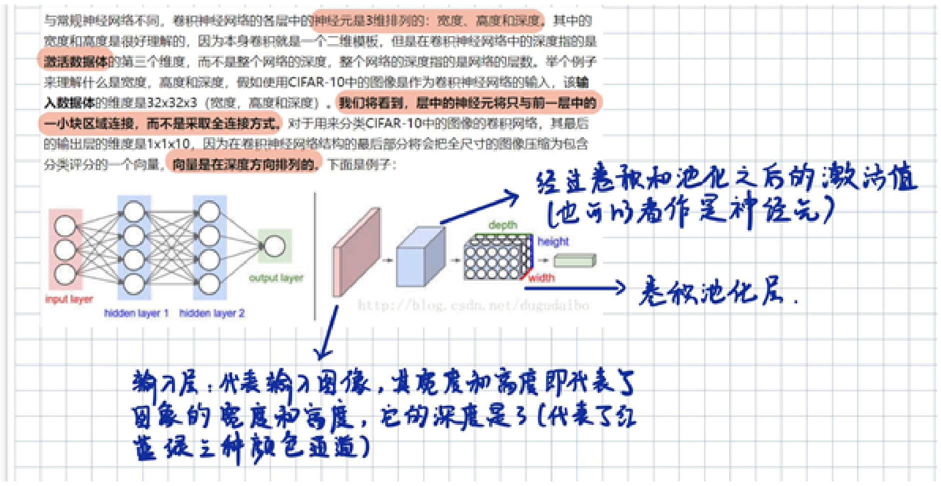
Advantages:

ReLU激活函数的提出 就是为了解决梯度消失问题，LSTMs也可用于解决梯度消失问题(但仅限于RNN模型)。ReLU的梯度只可以取两个值：0或1，当输入小于0时，梯度为0；当输入大于0时，梯度为1。好处就是：ReLU的梯度的连乘不会收敛到0，连乘的结果也只可以取两个值：0或1，如果值为1，梯度保持值不变进行前向传播；如果值为0 ,梯度从该位置停止前向传播。Sigmoid和ReLU函数对比如下：



2.3 CNN

Knowing the basic of the neural network, we can get into CNN:


与常规神经网络不同，卷积神经网络的各层中的神经元是3维排列的：宽度、高度和深度，其中的宽度和高度是很好理解的，因为本身卷积就是一个二维模板，但是在卷积神经网络中的深度指的是激活数据体的第三个维度，而不是整个网络的深度，整个网络的深度指的是网络的层数。举个例子来理解什么是宽度，高度和深度，假如使用CIFAR-10中的图像是作为卷积神经网络的输入，该**输入数据体**的维度是32x32x3（宽度，高度和深度）。**我们将看到，层中的神经元将只与前一层中的一小块区域连接，而不是采取全连接方式。**对于用来分类CIFAR-10中的图像的卷积网络，其最后的输出层的维度是1x1x10，因为在卷积神经网络结构的最后部分将会把全尺寸的图像压缩为包含分类评分的一个向量，**向量是在深度方向排列的。**下面是例子：

经过卷积和池化之后的激活值（也可以看作是神经元）

卷积池化层.

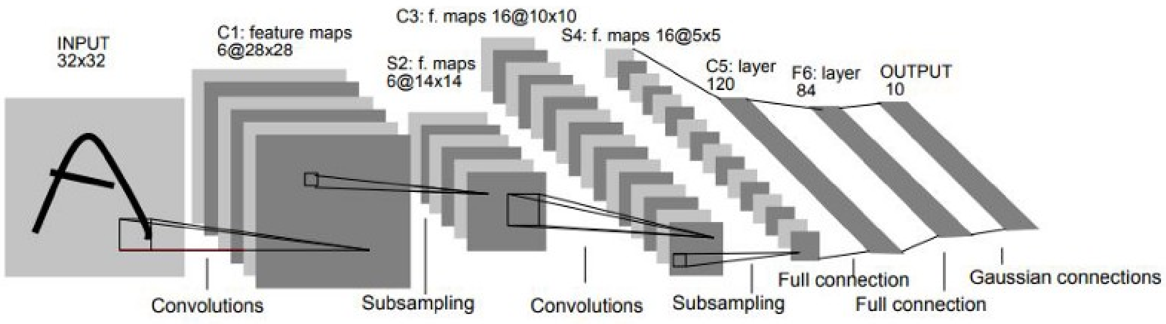
输入层：代表输入图像，其宽度和高度即代表了图像的宽度和高度，它的深度是3（代表了红蓝绿三种颜色通道）

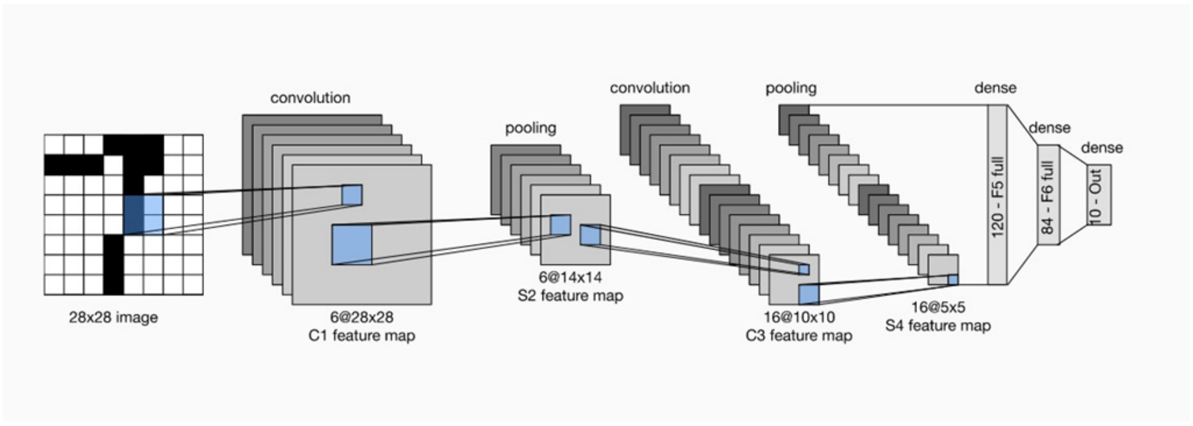
-
-
-

- Convolution Layer
- Pooling Layer
- Fully Connected Layer

2.4 LeNet

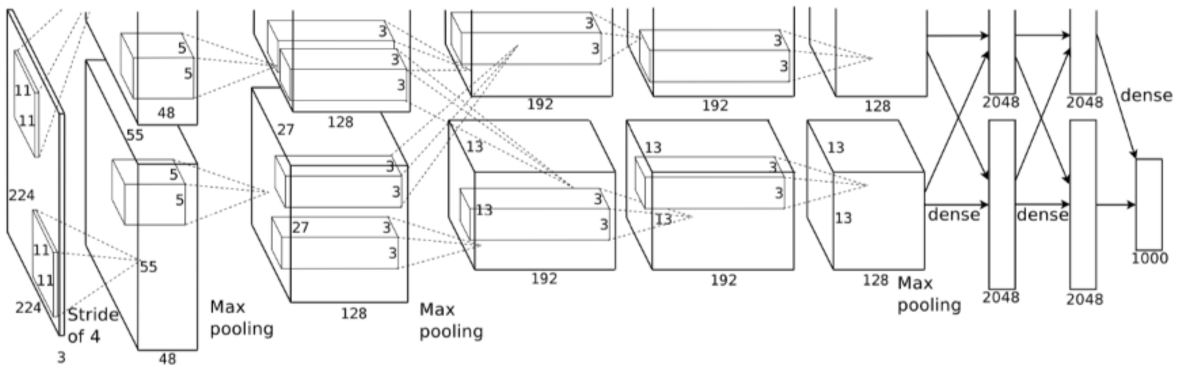
In 1989, Yann LeCun et al. at Bell Labs first applied the backpropagation algorithm to practical applications

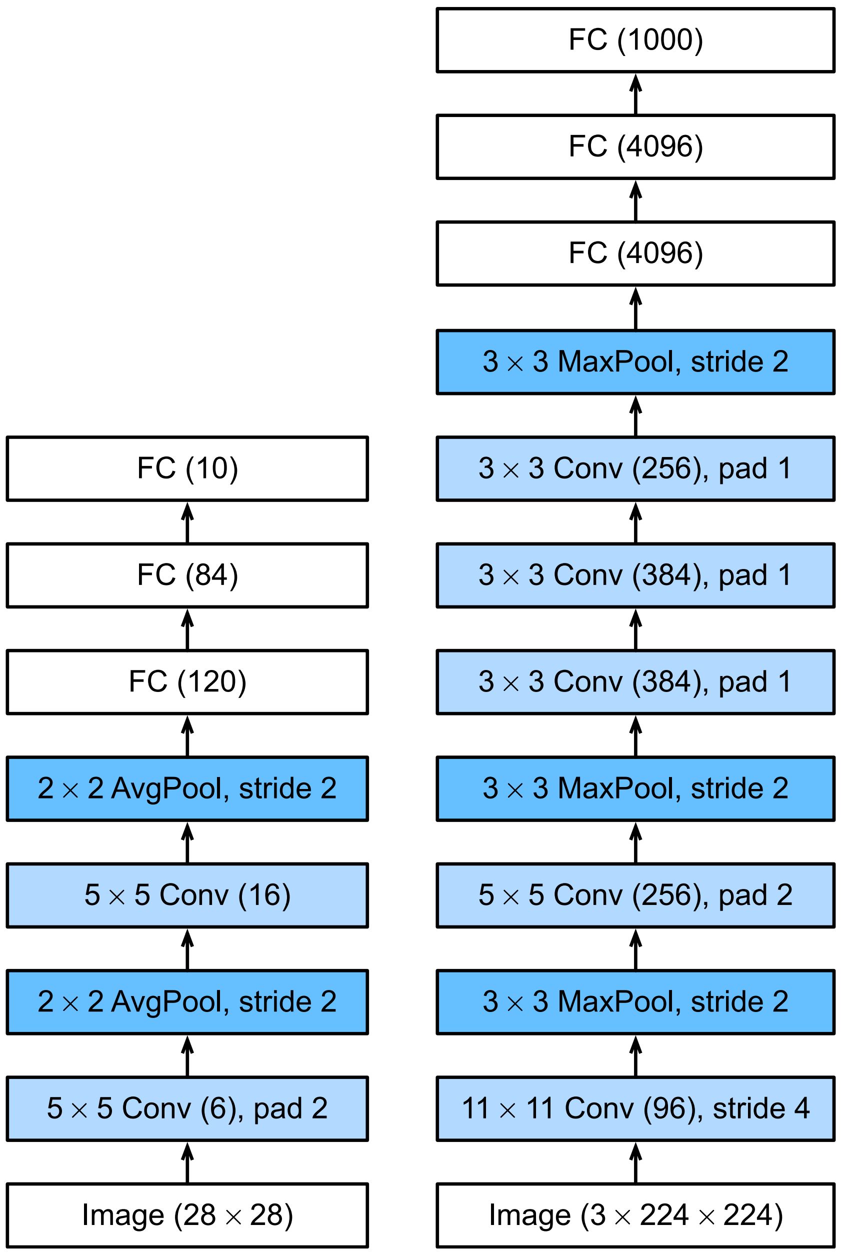




The basic units in each convolutional block are a convolutional layer, a sigmoid activation function, and a subsequent average pooling operation. Note that while ReLUs and max-pooling work better, these discoveries had not yet been made at the time.

2.5 AlexNet



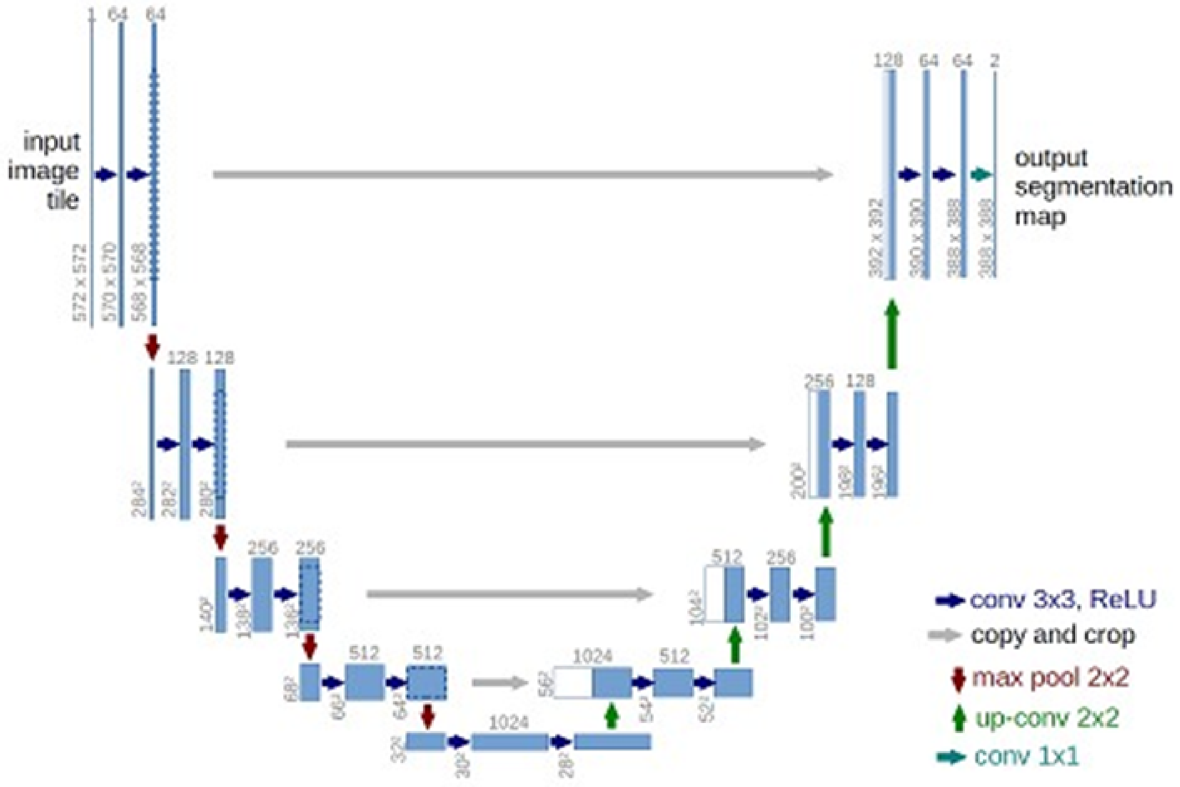


The architecture consists of eight layers: five convolutional layers and three fully-connected layers.

- **ReLU Nonlinearity**
- **Multiple GPUs**

3. Achievement

3.1 UNet



UNet is a type of convolutional neutral network that is initially used for medical image segmentation, which also
works well in file image binarization.

Structure of UNet:
[UNet]

Image binarization transfers a coloured image to a black-and-white one. The dataset we choose is [LS-HDIB], which contains contents, pages, and page textures. The training and testing images are generated by randomly combining each one of them.

Code to build UNet:

```
def unet_train(input_path, gt_path, output_path, weigths_path, start_epoch,
start_iter=0):
    batch_size = 8
    learning_rate = 0.0001
    epochs = 30

    train_data = DatasetLSHDIB(input_path + 'train', gt_path + 'train',
transform=transforms.ToTensor())
    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=False)

    val_data = DatasetLSHDIB(input_path + 'val', gt_path + 'val',
transform=transforms.ToTensor())
    val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=True)

    model = smp.Unet(in_channels=3, classes=1, activation="sigmoid")

    if start_iter != 0:
        model.load_state_dict(torch.load(weigths_path + 'iter%d_epoch%d.pth' %
(start_iter, start_epoch)))
    if start_iter == 0 and start_epoch != 0:
        wp_ = glob.glob(weigths_path + '*_epoch%d.pth' % start_epoch)
        wp_.sort()
        model.load_state_dict(torch.load(wp_[-1]))

    model = model.to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_loss = []
    val_loss = []

    for e in range(epochs):

        temp_loss = []

        if e == start_epoch and start_iter != 0:
            ary = np.load(output_path + 'epoch%d_loss_data.npy' % e)
            temp_loss = ary.tolist()

        model.train()

        if e >= start_epoch:
            model.train()
            for i, data in enumerate(tqdm(train_loader)):
                if e == start_epoch and i < start_iter:
                    continue
                x, y = data

                x = Variable(x).to(device)
                y = Variable(y).to(device)

                y_hat = model(x)
                # print(y_hat.shape)
                # print(y.shape)
                loss = rgb_bcel(y_hat, y)
                # loss = gray_mse(y_hat, y)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()
                print("Epoch %d Batch %d Loss : " % (e, i), loss.item())
                temp_loss.append(loss.item())

                if i % 100 == 0:
                    print("Saving an image")
                    print("Epoch ", e)
                    print("Loss ", loss.item())
                    out_img = y_hat.cpu().data
                    # vutils.save_image(y,output_path+'GT_%d_epoch_%d.jpg'%
(i,e),normalize=False)
                    # vutils.save_image(x,output_path+'Input_%d_epoch_%d.jpg'%
(i,e),normalize=False)
                    vutils.save_image(y, output_path + 'Train_GT_%d.jpg' % i,
normalize=False)
                    vutils.save_image(x, output_path + 'Train_Input_%d.jpg' % i,
normalize=False)
```

```
                    vutils.save_image(out_img, output_path +
'Train_Pred_%d_epoch_%d.jpg' % (i, e), normalize=False)
                    torch.save(model.state_dict(), weigths_path +
'iter%d_epoch%d.pth' % (i, e))
                    title = 'PageNet Epoch%d loss per iteration'%e
                    plt.figure()
                    plt.plot(temp_loss, label="training loss")
                    plt.title(title)
                    plt.legend()
                    plt.savefig(output_path+'epoch%d_loss_data.png'%e)
                    np.save(output_path + 'epoch%d_loss_data.npy' % e,
temp_loss)
            if i > 400:
                exit(-1)
            train_loss.append(np.sum(temp_loss) / len(temp_loss))

            temp_loss = []
            model.eval()
            for i, data in enumerate(val_loader):
                x, y = data

                x = Variable(x).to(device)
                y = Variable(y).to(device)

                y_hat = model(x)
                loss = rgb_bcel(y_hat, y) * 100

                print("Epoch %d Batch %d Validation Loss : " % (e, i),
loss.item())
                temp_loss.append(loss.item())
                if i % 50 == 0:
                    out_img = y_hat.cpu().data
                    vutils.save_image(y, output_path + 'Val_GT_%d.jpg' % i,
normalize=False)
                    vutils.save_image(x, output_path + 'Val_Input_%d.jpg' % i,
normalize=False)
                    vutils.save_image(out_img, output_path +
'Val_Pred_%d_epoch_%d.jpg' % (i, e), normalize=False)
                plt.plot(temp_loss, label="validation loss")
                plt.title(title)
                plt.legend()
                plt.savefig(output_path+'lossData.png')
            val_loss.append(np.sum(temp_loss) / len(temp_loss))
```

3.1.1 Training model

The first three epoches:

```
Device used :  cpu
Number of images =  216
Number of images =  216
  0%|          | 0/27 [00:00<?, ?it/s]Epoch 0 Batch 0 Loss :  0.7202784419059753
Saving an image
Epoch  0
Loss  0.7202784419059753
```

```
  7%|▋         | 2/27 [00:15<03:12,  7.72s/it]Epoch 0 Batch 1 Loss :
0.7037879824638367
 11%|█         | 3/27 [00:22<03:01,  7.57s/it]Epoch 0 Batch 2 Loss :
0.6865094900131226
Epoch 0 Batch 3 Loss :  0.6792236566543579
 19%|█▊        | 5/27 [00:37<02:46,  7.56s/it]Epoch 0 Batch 4 Loss :
0.6579667329788208
Epoch 0 Batch 5 Loss :  0.6523716449737549
 26%|██▌       | 7/27 [00:55<02:45,  8.25s/it]Epoch 0 Batch 6 Loss :
0.6411563158035278
 30%|██▉       | 8/27 [01:04<02:40,  8.47s/it]Epoch 0 Batch 7 Loss :
0.629039466381073
Epoch 0 Batch 8 Loss :  0.6125644445419312
 37%|███▋      | 10/27 [01:22<02:30,  8.83s/it]Epoch 0 Batch 9 Loss :
0.6092973351478577
Epoch 0 Batch 10 Loss :  0.5922699570655823
 44%|████▍     | 12/27 [01:40<02:14,  8.93s/it]Epoch 0 Batch 11 Loss :
0.5777613520622253
Epoch 0 Batch 12 Loss :  0.5834792256355286
 52%|█████▏    | 14/27 [01:59<01:57,  9.04s/it]Epoch 0 Batch 13 Loss :
0.5824288129806519
 56%|█████▌    | 15/27 [02:08<01:48,  9.04s/it]Epoch 0 Batch 14 Loss :
0.5581243634223938
 59%|█████▉    | 16/27 [02:17<01:40,  9.10s/it]Epoch 0 Batch 15 Loss :
0.554010808467865
Epoch 0 Batch 16 Loss :  0.5402776598930359
 67%|██████▋   | 18/27 [02:35<01:21,  9.08s/it]Epoch 0 Batch 17 Loss :
0.5402215123176575
Epoch 0 Batch 18 Loss :  0.523686408996582
 74%|███████▍  | 20/27 [02:53<01:03,  9.02s/it]Epoch 0 Batch 19 Loss :
0.5346329808235168
Epoch 0 Batch 20 Loss :  0.5300207734107971
 81%|████████▏ | 22/27 [03:14<00:48,  9.69s/it]Epoch 0 Batch 21 Loss :
0.5126689672470093
 85%|████████▌ | 23/27 [03:23<00:38,  9.59s/it]Epoch 0 Batch 22 Loss :
0.49661099910736084
Epoch 0 Batch 23 Loss :  0.49294018745422363
 93%|█████████▎| 25/27 [03:42<00:18,  9.44s/it]Epoch 0 Batch 24 Loss :
0.49078550934791565
 96%|█████████▋| 26/27 [03:51<00:09,  9.34s/it]Epoch 0 Batch 25 Loss :
0.49846503138542175
Epoch 0 Batch 26 Loss :  0.4835210144519806
100%|██████████| 27/27 [04:00<00:00,  8.92s/it]
Epoch 0 Batch 0 Validation Loss :  49.36328887939453
Epoch 0 Batch 1 Validation Loss :  50.74656677246094
Epoch 0 Batch 2 Validation Loss :  51.52290344238281
Epoch 0 Batch 3 Validation Loss :  49.38803482055664
Epoch 0 Batch 4 Validation Loss :  49.87494659423828
Epoch 0 Batch 5 Validation Loss :  49.877593994140625
Epoch 0 Batch 6 Validation Loss :  49.574588775634766
Epoch 0 Batch 7 Validation Loss :  50.73482894897461
Epoch 0 Batch 8 Validation Loss :  50.07638168334961
Epoch 0 Batch 9 Validation Loss :  50.96192169189453
Epoch 0 Batch 10 Validation Loss :  48.17668151855469
Epoch 0 Batch 11 Validation Loss :  51.10974884033203
Epoch 0 Batch 12 Validation Loss :  49.32221984863281
Epoch 0 Batch 13 Validation Loss :  50.61429214477539
Epoch 0 Batch 14 Validation Loss :  50.25292205810547
Epoch 0 Batch 15 Validation Loss :  50.896263122558594
Epoch 0 Batch 16 Validation Loss :  50.36189651489258
Epoch 0 Batch 17 Validation Loss :  49.19540786743164
Epoch 0 Batch 18 Validation Loss :  48.65625
Epoch 0 Batch 19 Validation Loss :  50.985836029052734
Epoch 0 Batch 20 Validation Loss :  52.609466552734375
Epoch 0 Batch 21 Validation Loss :  50.24110794067383
Epoch 0 Batch 22 Validation Loss :  50.63648986816406
Epoch 0 Batch 23 Validation Loss :  48.97722625732422
Epoch 0 Batch 24 Validation Loss :  50.76877975463867
Epoch 0 Batch 25 Validation Loss :  49.205848693847656
Epoch 0 Batch 26 Validation Loss :  52.06869125366211
  0%|          | 0/27 [00:00<?, ?it/s]Epoch 1 Batch 0 Loss :
0.48133009672164917
Saving an image
Epoch  1
Loss  0.48133009672164917
  7%|▋         | 2/27 [00:21<04:23, 10.53s/it]Epoch 1 Batch 1 Loss :
0.46848320960998535
 11%|█         | 3/27 [00:31<04:09, 10.39s/it]Epoch 1 Batch 2 Loss :
0.46513718366622925
 15%|█▍        | 4/27 [00:41<03:54, 10.21s/it]Epoch 1 Batch 3 Loss :
0.466458261013031
 19%|█▊        | 5/27 [00:51<03:44, 10.19s/it]Epoch 1 Batch 4 Loss :
0.45720013976097107
 22%|██▏       | 6/27 [01:01<03:33, 10.16s/it]Epoch 1 Batch 5 Loss :
0.46118322014808655
Epoch 1 Batch 6 Loss :  0.4473888874053955
 30%|██▉       | 8/27 [01:21<03:12, 10.15s/it]Epoch 1 Batch 7 Loss :
0.4446142613887787
 33%|███▎      | 9/27 [01:32<03:02, 10.14s/it]Epoch 1 Batch 8 Loss :
0.43418043851852417
Epoch 1 Batch 9 Loss :  0.4434989392757416
 41%|████      | 11/27 [01:51<02:37,  9.86s/it]Epoch 1 Batch 10 Loss :
0.42706501483917236
 44%|████▍     | 12/27 [02:01<02:28,  9.89s/it]Epoch 1 Batch 11 Loss :
0.42876192927360535
Epoch 1 Batch 12 Loss :  0.4162820875644684
 52%|█████▏    | 14/27 [02:22<02:15, 10.39s/it]Epoch 1 Batch 13 Loss :
0.4169921576976776
 56%|█████▌    | 15/27 [02:33<02:05, 10.43s/it]Epoch 1 Batch 14 Loss :
0.4106666147708893
 59%|█████▉    | 16/27 [02:43<01:52, 10.26s/it]Epoch 1 Batch 15 Loss :
0.4062899053096771
 63%|██████▎   | 17/27 [02:52<01:40, 10.10s/it]Epoch 1 Batch 16 Loss :
0.40352097153663635
Epoch 1 Batch 17 Loss :  0.4087482988834381
 70%|███████   | 19/27 [03:12<01:19,  9.97s/it]Epoch 1 Batch 18 Loss :
0.3991386592388153
 74%|███████▍  | 20/27 [03:22<01:09,  9.90s/it]Epoch 1 Batch 19 Loss :
0.39671698212623596
Epoch 1 Batch 20 Loss :  0.40606001019477844
```

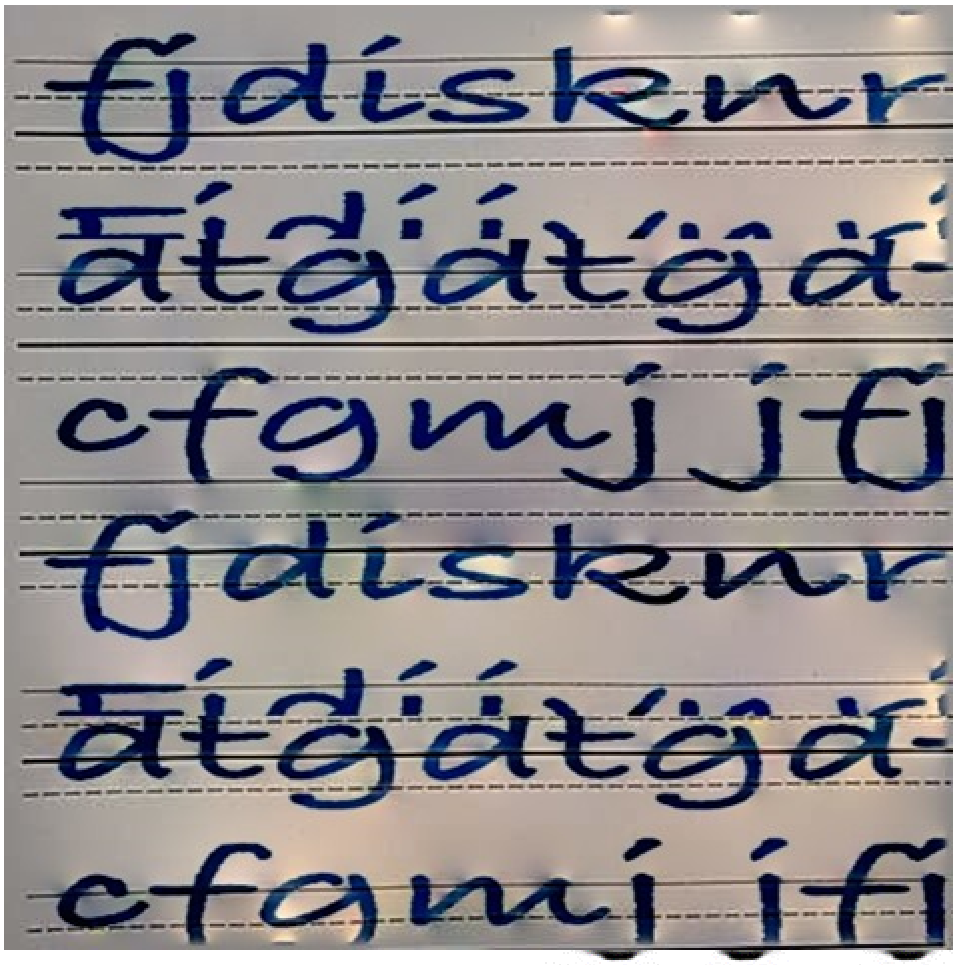
```
 81%|████████▏ | 22/27 [03:42<00:49,  9.99s/it]Epoch 1 Batch 21 Loss :
0.3913322389125824
 85%|████████▌ | 23/27 [03:52<00:39,  9.95s/it]Epoch 1 Batch 22 Loss :
0.3843064308166504
 89%|████████▉ | 24/27 [04:02<00:29,  9.96s/it]Epoch 1 Batch 23 Loss :
0.373668372631073
 93%|█████████▎| 25/27 [04:12<00:19,  9.95s/it]Epoch 1 Batch 24 Loss :
0.37659576535224915
 96%|█████████▋| 26/27 [04:22<00:09,  9.91s/it]Epoch 1 Batch 25 Loss :
0.37711262702941895
Epoch 1 Batch 26 Loss :  0.37987780570983887
100%|██████████| 27/27 [04:32<00:00, 10.08s/it]
Epoch 1 Batch 0 Validation Loss :  38.07993698120117
Epoch 1 Batch 1 Validation Loss :  33.7322998046875
Epoch 1 Batch 2 Validation Loss :  33.356075286865234
Epoch 1 Batch 3 Validation Loss :  34.798465728759766
Epoch 1 Batch 4 Validation Loss :  36.15454864501953
Epoch 1 Batch 5 Validation Loss :  36.027008056640625
Epoch 1 Batch 6 Validation Loss :  33.65613555908203
Epoch 1 Batch 7 Validation Loss :  34.29631042480469
Epoch 1 Batch 8 Validation Loss :  33.53697204589844
Epoch 1 Batch 9 Validation Loss :  34.08867263793945
Epoch 1 Batch 10 Validation Loss :  34.55353546142578
Epoch 1 Batch 11 Validation Loss :  31.74789047241211
Epoch 1 Batch 12 Validation Loss :  36.00352478027344
Epoch 1 Batch 13 Validation Loss :  34.551517486572266
Epoch 1 Batch 14 Validation Loss :  36.08701705932617
Epoch 1 Batch 15 Validation Loss :  36.13054275512695
Epoch 1 Batch 16 Validation Loss :  33.72124481201172
Epoch 1 Batch 17 Validation Loss :  33.64496612548828
Epoch 1 Batch 18 Validation Loss :  34.875423431396484
Epoch 1 Batch 19 Validation Loss :  37.14402770996094
Epoch 1 Batch 20 Validation Loss :  34.35379409790039
Epoch 1 Batch 21 Validation Loss :  34.33844757080078
Epoch 1 Batch 22 Validation Loss :  34.76601791381836
Epoch 1 Batch 23 Validation Loss :  37.82840347290039
Epoch 1 Batch 24 Validation Loss :  31.115049362182617
Epoch 1 Batch 25 Validation Loss :  33.97903823852539
Epoch 1 Batch 26 Validation Loss :  34.77248764038086
  0%|          | 0/27 [00:00<?, ?it/s]Epoch 2 Batch 0 Loss :
0.37737059593200684
Saving an image
Epoch  2
Loss  0.37737059593200684
  7%|▋         | 2/27 [00:21<04:23, 10.54s/it]Epoch 2 Batch 1 Loss :
0.36003533005714417
 11%|█         | 3/27 [00:31<04:09, 10.39s/it]Epoch 2 Batch 2 Loss :
0.3580823540687561
Epoch 2 Batch 3 Loss :  0.3620556890964508
 19%|█▊        | 5/27 [00:51<03:45, 10.25s/it]Epoch 2 Batch 4 Loss :
0.3515089154243469
Epoch 2 Batch 5 Loss :  0.3650375008583069
 26%|██▌       | 7/27 [01:11<03:20, 10.01s/it]Epoch 2 Batch 6 Loss :
0.3604469895362854
```

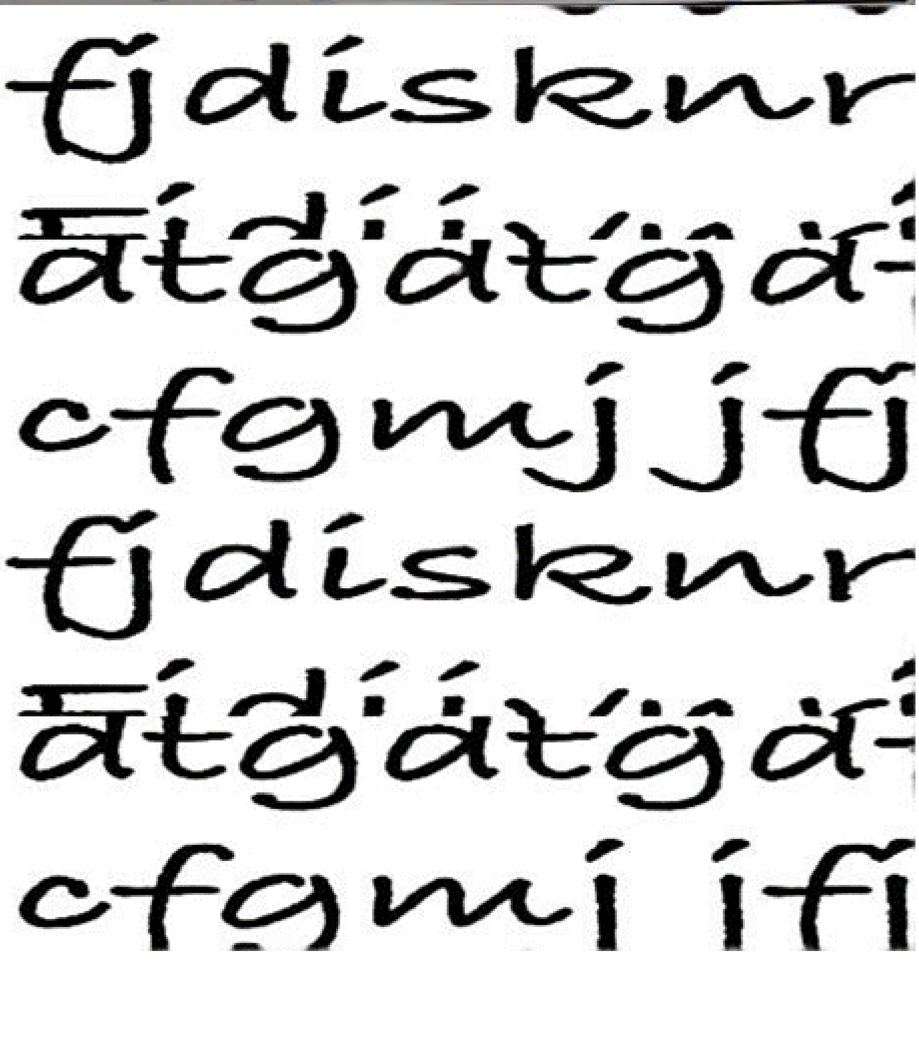
```
 30%|███        | 8/27 [01:21<03:08,  9.90s/it]Epoch 2 Batch 7 Loss :
0.3540363907814026
Epoch 2 Batch 8 Loss :  0.3428625166416168
 37%|███▋       | 10/27 [01:40<02:48,  9.91s/it]Epoch 2 Batch 9 Loss :
0.3605545461177826
Epoch 2 Batch 10 Loss :  0.3417091965675354
 44%|████▍      | 12/27 [02:00<02:28,  9.91s/it]Epoch 2 Batch 11 Loss :
0.35411345958709717
Epoch 2 Batch 12 Loss :  0.3371458351612091
 52%|█████▏     | 14/27 [02:19<02:05,  9.68s/it]Epoch 2 Batch 13 Loss :
0.33476054668426514
 56%|█████▌     | 15/27 [02:29<01:55,  9.63s/it]Epoch 2 Batch 14 Loss :
0.33155402541160583
Epoch 2 Batch 15 Loss :  0.33105337619781494
 63%|██████▎    | 17/27 [02:48<01:36,  9.61s/it]Epoch 2 Batch 16 Loss :
0.3474927842617035
 67%|██████▋    | 18/27 [02:58<01:26,  9.63s/it]Epoch 2 Batch 17 Loss :
0.33970892429351807
 70%|███████    | 19/27 [03:07<01:16,  9.56s/it]Epoch 2 Batch 18 Loss :
0.33589816093444824
 74%|███████▍   | 20/27 [03:17<01:07,  9.59s/it]Epoch 2 Batch 19 Loss :
0.3282562494277954
Epoch 2 Batch 20 Loss :  0.3411406874656677
 81%|████████▏  | 22/27 [03:36<00:47,  9.58s/it]Epoch 2 Batch 21 Loss :
0.32979950308799744
Epoch 2 Batch 22 Loss :  0.3255186080932617
 89%|████████▉  | 24/27 [03:55<00:28,  9.60s/it]Epoch 2 Batch 23 Loss :
0.3110479712486267
Epoch 2 Batch 24 Loss :  0.3166217803955078
 96%|█████████▋| 26/27 [04:14<00:09,  9.59s/it]Epoch 2 Batch 25 Loss :
0.3211003839969635
100%|██████████| 27/27 [04:24<00:00,  9.79s/it]
Epoch 2 Batch 26 Loss :  0.3244677782058716
Epoch 2 Batch 0 Validation Loss :  28.00088882446289
Epoch 2 Batch 1 Validation Loss :  28.284008026123047
Epoch 2 Batch 2 Validation Loss :  30.80017852783203
Epoch 2 Batch 3 Validation Loss :  29.230005264282227
Epoch 2 Batch 4 Validation Loss :  27.607574462890625
Epoch 2 Batch 5 Validation Loss :  31.210735321044922
Epoch 2 Batch 6 Validation Loss :  31.463428497314453
Epoch 2 Batch 7 Validation Loss :  29.219890594482422
Epoch 2 Batch 8 Validation Loss :  29.83441925048828
Epoch 2 Batch 9 Validation Loss :  30.33510398864746
Epoch 2 Batch 10 Validation Loss :  28.905136108398438
Epoch 2 Batch 11 Validation Loss :  31.219404220581055
Epoch 2 Batch 12 Validation Loss :  30.807462692260742
Epoch 2 Batch 13 Validation Loss :  29.700315475463867
Epoch 2 Batch 14 Validation Loss :  29.62126350402832
Epoch 2 Batch 15 Validation Loss :  29.700634002685547
Epoch 2 Batch 16 Validation Loss :  30.272695541381836
Epoch 2 Batch 17 Validation Loss :  30.347299575805664
Epoch 2 Batch 18 Validation Loss :  28.930389404296875
Epoch 2 Batch 19 Validation Loss :  29.72594451904297
Epoch 2 Batch 20 Validation Loss :  31.67351531982422
Epoch 2 Batch 21 Validation Loss :  30.961544036865234
Epoch 2 Batch 22 Validation Loss :  30.54358673095703
Epoch 2 Batch 23 Validation Loss :  30.217023849487305
Epoch 2 Batch 24 Validation Loss :  30.4843692779541
Epoch 2 Batch 25 Validation Loss :  29.400840759277344
Epoch 2 Batch 26 Validation Loss :  29.550765991210938
```

The training need 30 epoches. At epoch 30, the training loss is about 0.07, and validation loss is about 12.

3.1.2 Application

For each epoch, we get a Weights-UNet file. The Weights-UNet file that has least training + testing loss is
our UNet-best-weights file.


fjdisknr
ātgágátgá-
cfgmjjfi
fjdisknr
ātgágátgá-
cfgmjifi


fjdisknr
ātgágátgá-
cfgmjjfi
fjdisknr
ātgágátgá-
cfgmjifi

3.2 KNN

To begin our project, wo decide to start from some easy algorithms like KNN.

The data set we choose is EMNIST. The EMNIST dataset is a set of handwritten character digits derived from the NIST Special Database 19 and converted to a 28x28 pixel image format and dataset structure that directly matches the MNIST dataset.

Codes to build KNN:

```
def knn(K):
    train_x, train_y, test_x, test_y = load_data()
    cnt = 0
    for i in range(len(test_x)):
        # print(i)
        x = test_x[i]
        y = test_y[i]
        vec = get_vec(K, x, train_x, train_y)
        weight = []  # 权重与序号
        sum_distance = 0.0
        for j in range(K):
            sum_distance += vec[j][0]  # 计算前K个距离的和
        for j in range(K):
            weight.append([1 - vec[j][0] / sum_distance, vec[j][1]])  # 权重+序号
        # 将相同序号的加起来
        num = []  # 统计有哪些序号
        for j in range(K):
            num.append(weight[j][1])
        num = list(set(num))  # 去重

        final_res = []
        for j in range(len(num)):
            res = 0.0
            for k in range(len(weight)):
                if weight[k][1] == num[j]:  # 前K个标签一样的样本权值加起来
                    res += weight[k][0]
            final_res.append([res, num[j]])

        final_res = sorted(final_res, key=(lambda e: e[0]), reverse=True)  # 按照
权重从大到小排序

        if y == final_res[0][1]:
            cnt = cnt + 1
        print(y, final_res[0][1],i)

    print('accuracy:', cnt / len(test_x))
```

3.3 LeNet

The second algorithm we choose is LeNet. This is a little bit complex but it is still the simplest CNN. Due to the limitation of the original LeNet, we change some structures into the popular ones. For example, we change the activation function from Sigmoid into ReLU and change the average pooling into max pooling.

Codes to build a CNN:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(  # input shape (1, 28, 28)
            nn.Conv2d(
                in_channels=1,  # 输入通道数
                out_channels=16,  # 输出通道数
                kernel_size=5,   # 卷积核大小
                stride=1,  #卷积步数
                padding=2,  # 如果想要 con2d 出来的图片长宽没有变化,
                            # padding=(kernel_size-1)/2 当 stride=1
            ),  # output shape (16, 28, 28)
            nn.ReLU(),  # activation
            nn.MaxPool2d(kernel_size=2),  # 在 2x2 空间里向下采样, output shape
(16, 14, 14)
        )
        self.conv2 = nn.Sequential(  # input shape (16, 14, 14)
            nn.Conv2d(16, 32, 5, 1, 2),  # output shape (32, 14, 14)
            nn.ReLU(),  # activation
            nn.MaxPool2d(2),  # output shape (32, 7, 7)
        )
        self.out = nn.Linear(32 * 7 * 7, 37)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)  # 展平多维的卷积图成 (batch_size, 32 * 7 * 7)
        output = self.out(x)
        return output
```

3.4 AlexNet

The last algorithm we use is the famous AlexNet.

The original AlexNet uses two GPUs to work together. But with the technology developing so quickly, one GPU is enough for us. So we cut the structure to half.

Codes to build Alexnet:

```
class AlexNet(nn.Module):
    def __init__(self, num_classes=1000, init_weights=False):
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(  # 打包
            nn.Conv2d(3, 48, kernel_size=11, stride=4, padding=2),  # input[3,
224, 224]  output[48, 55, 55] 自动舍去小数点后
            nn.ReLU(inplace=True),  # inplace 可以载入更大模型
            nn.MaxPool2d(kernel_size=3, stride=2),  # output[48, 27, 27]
kernel_num为原论文一半
            nn.Conv2d(48, 128, kernel_size=5, padding=2),  # output[128, 27, 27]
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),  # output[128, 13, 13]
            nn.Conv2d(128, 192, kernel_size=3, padding=1),  # output[192, 13,
13]
            nn.ReLU(inplace=True),
            nn.Conv2d(192, 192, kernel_size=3, padding=1),  # output[192, 13,
13]
            nn.ReLU(inplace=True),
            nn.Conv2d(192, 128, kernel_size=3, padding=1),  # output[128, 13,
13]
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),  # output[128, 6, 6]
        )
        self.classifier = nn.Sequential(
            nn.Dropout(p=0.5),
            # 全链接
            nn.Linear(128 * 6 * 6, 2048),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),
            nn.Linear(2048, 2048),
            nn.ReLU(inplace=True),
            nn.Linear(2048, num_classes),
        )
        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, start_dim=1)  # 展平或者view()
        x = self.classifier(x)
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)  # 正态分布赋值
                nn.init.constant_(m.bias, 0)
```

3.5 Comparison

EMNIST:

```
train_data = torchvision.datasets.EMNIST(
    root='./data',
    train=True,
    transform=torchvision.transforms.ToTensor(),
    download = True,
    split = 'letters'
)
# EMNIST 手写字母 测试集
test_data = torchvision.datasets.EMNIST(
    root='./data',
    train=False,
    transform=torchvision.transforms.ToTensor(),
    download=True,
    split = 'letters'
)
```

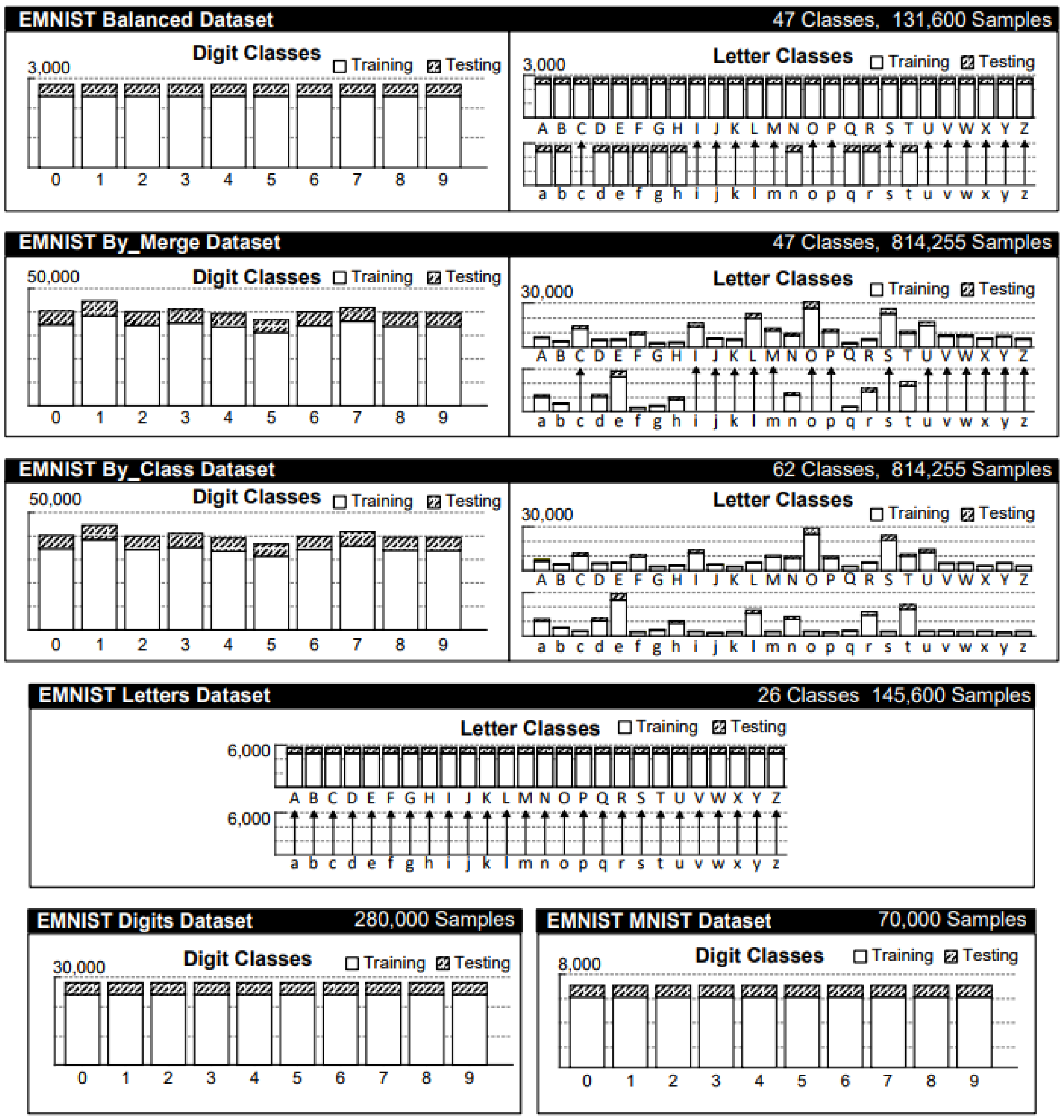
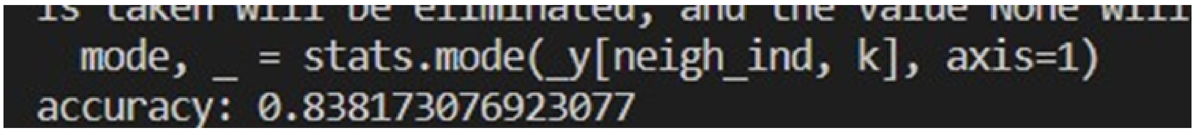


Fig. 2. **Visual breakdown of the EMNIST datasets.** The class breakdown, structure and splits of the various datasets in the EMNIST dataset are shown. Each dataset contains handwritten digits, handwritten letters or a combination of both. The number of samples in each class is shown for each dataset and highlights the large variation in the number of samples in the unbalanced datasets. The training and testing split for each class is also shown using either solid or hatched portions of the bar graphs. In the datasets that contain merged classes, a vertical arrow is used to denote the class into which the lowercase letter is merged.

We use Letter Classes and By_Classes here.

3.5.1 KNN



```
  is taken will be eliminated, and the value None will
     mode, _ = stats.mode(_y[neigh_ind, k], axis=1)
  accuracy: 0.838173076923077
```

letters:Accuracy:83.817%

3.5.2 CNN

letters:Accuracy:91%

```
Epoch:  0 | train loss: 3.6121 | test accuracy: 0.08
Epoch:  0 | train loss: 1.6627 | test accuracy: 0.50
Epoch:  0 | train loss: 1.1810 | test accuracy: 0.65
```

```
Epoch:  0 | train loss: 0.9111 | test accuracy: 0.64
Epoch:  0 | train loss: 0.6663 | test accuracy: 0.72
Epoch:  0 | train loss: 0.7040 | test accuracy: 0.75
Epoch:  0 | train loss: 0.9459 | test accuracy: 0.79
Epoch:  0 | train loss: 0.7983 | test accuracy: 0.81
Epoch:  0 | train loss: 0.4633 | test accuracy: 0.78
Epoch:  0 | train loss: 0.4120 | test accuracy: 0.80
...
Epoch:  0 | train loss: 0.2542 | test accuracy: 0.86
Epoch:  0 | train loss: 0.3830 | test accuracy: 0.85
Epoch:  0 | train loss: 0.4844 | test accuracy: 0.86
Epoch:  0 | train loss: 0.2865 | test accuracy: 0.88
Epoch:  0 | train loss: 0.4786 | test accuracy: 0.86
Epoch:  0 | train loss: 0.3413 | test accuracy: 0.87
Epoch:  0 | train loss: 0.3735 | test accuracy: 0.87
Epoch:  0 | train loss: 0.5593 | test accuracy: 0.86
Epoch:  0 | train loss: 0.2307 | test accuracy: 0.87
Epoch:  0 | train loss: 0.5639 | test accuracy: 0.85
Epoch:  0 | train loss: 0.3532 | test accuracy: 0.87
Epoch:  0 | train loss: 0.2593 | test accuracy: 0.87
Epoch:  0 | train loss: 0.9487 | test accuracy: 0.87
Epoch:  0 | train loss: 0.2290 | test accuracy: 0.88
Epoch:  0 | train loss: 0.2218 | test accuracy: 0.88
Epoch:  0 | train loss: 0.2571 | test accuracy: 0.87
Epoch:  0 | train loss: 0.2921 | test accuracy: 0.88
Epoch:  0 | train loss: 0.2959 | test accuracy: 0.87
Epoch:  0 | train loss: 0.1770 | test accuracy: 0.87
Epoch:  0 | train loss: 0.4298 | test accuracy: 0.88
Epoch:  0 | train loss: 0.1231 | test accuracy: 0.88
Epoch:  0 | train loss: 0.1818 | test accuracy: 0.89
Epoch:  0 | train loss: 0.2442 | test accuracy: 0.88
Epoch:  0 | train loss: 0.1373 | test accuracy: 0.88
Epoch:  0 | train loss: 0.2890 | test accuracy: 0.88
Epoch:  0 | train loss: 0.1318 | test accuracy: 0.87
Epoch:  0 | train loss: 0.2625 | test accuracy: 0.87
Epoch:  0 | train loss: 0.2961 | test accuracy: 0.88
Epoch:  1 | train loss: 0.2997 | test accuracy: 0.89
Epoch:  1 | train loss: 0.4059 | test accuracy: 0.90
Epoch:  1 | train loss: 0.2920 | test accuracy: 0.88
Epoch:  1 | train loss: 0.3765 | test accuracy: 0.89
Epoch:  1 | train loss: 0.3220 | test accuracy: 0.89
Epoch:  1 | train loss: 0.1546 | test accuracy: 0.90
Epoch:  2 | train loss: 0.1815 | test accuracy: 0.91
Epoch:  2 | train loss: 0.1485 | test accuracy: 0.90
Epoch:  2 | train loss: 0.2233 | test accuracy: 0.90
Epoch:  2 | train loss: 0.2066 | test accuracy: 0.89
Epoch:  2 | train loss: 0.1848 | test accuracy: 0.91
Epoch:  2 | train loss: 0.1072 | test accuracy: 0.90
Epoch:  2 | train loss: 0.2494 | test accuracy: 0.90
...
Epoch:  9 | train loss: 0.0770 | test accuracy: 0.90
Epoch:  9 | train loss: 0.2014 | test accuracy: 0.90
Epoch:  9 | train loss: 0.0926 | test accuracy: 0.91
Epoch:  9 | train loss: 0.1080 | test accuracy: 0.91
Epoch:  9 | train loss: 0.0501 | test accuracy: 0.91
```

```
Epoch:  9 | train loss: 0.1451 | test accuracy: 0.91
```

CNN prediction accuracy growth is strange, and it's not very high based on letters.

3.5.3 Alexnet

```
Training the model...



Epoch:  12%|█▏        | 1/8 [07:29<52:25, 449.41s/epoch]
Epoch [0], last_lr: 0.00396, train_loss: 0.6916, val_loss: 0.4611, val_acc:
0.8368



Epoch:  25%|██▌       | 2/8 [15:04<45:18, 453.02s/epoch]
Epoch [1], last_lr: 0.00936, train_loss: 0.6260, val_loss: 0.5618, val_acc:
0.8115



Epoch:  38%|███▊      | 3/8 [22:36<37:40, 452.20s/epoch]
Epoch [2], last_lr: 0.00972, train_loss: 0.6582, val_loss: 0.5053, val_acc:
0.8225



Epoch:  50%|█████     | 4/8 [30:08<30:09, 452.28s/epoch]
Epoch [3], last_lr: 0.00812, train_loss: 0.5952, val_loss: 0.4679, val_acc:
0.8312



Epoch:  62%|██████▎   | 5/8 [37:36<22:32, 450.72s/epoch]
Epoch [4], last_lr: 0.00556, train_loss: 0.5351, val_loss: 0.4218, val_acc:
0.8487



Epoch:  75%|███████▌  | 6/8 [45:02<14:58, 449.19s/epoch]
Epoch [5], last_lr: 0.00283, train_loss: 0.4686, val_loss: 0.3800, val_acc:
0.8589



Epoch:  88%|████████▊ | 7/8 [52:29<07:28, 448.54s/epoch]
Epoch [6], last_lr: 0.00077, train_loss: 0.4110, val_loss: 0.3509, val_acc:
0.8690

Epoch: 100%|██████████| 8/8 [59:59<00:00, 449.94s/epoch]
Epoch [7], last_lr: 0.00000, train_loss: 0.3716, val_loss: 0.3389, val_acc:
0.8728
```

byclass:Accuracy:87.28%

As the results shown above, the accuracy based on CNN with letters is not far higher than Alexnet with by_class, besides, the CNN prediction accuracy growth is strange, so finally I choose Alexnet as the model we use to predict handwritten letters.

3.6 Application

After comparison, we conclude that AlexNet did a better job. So we decide to make a visualizing program to implement this wonderful network.

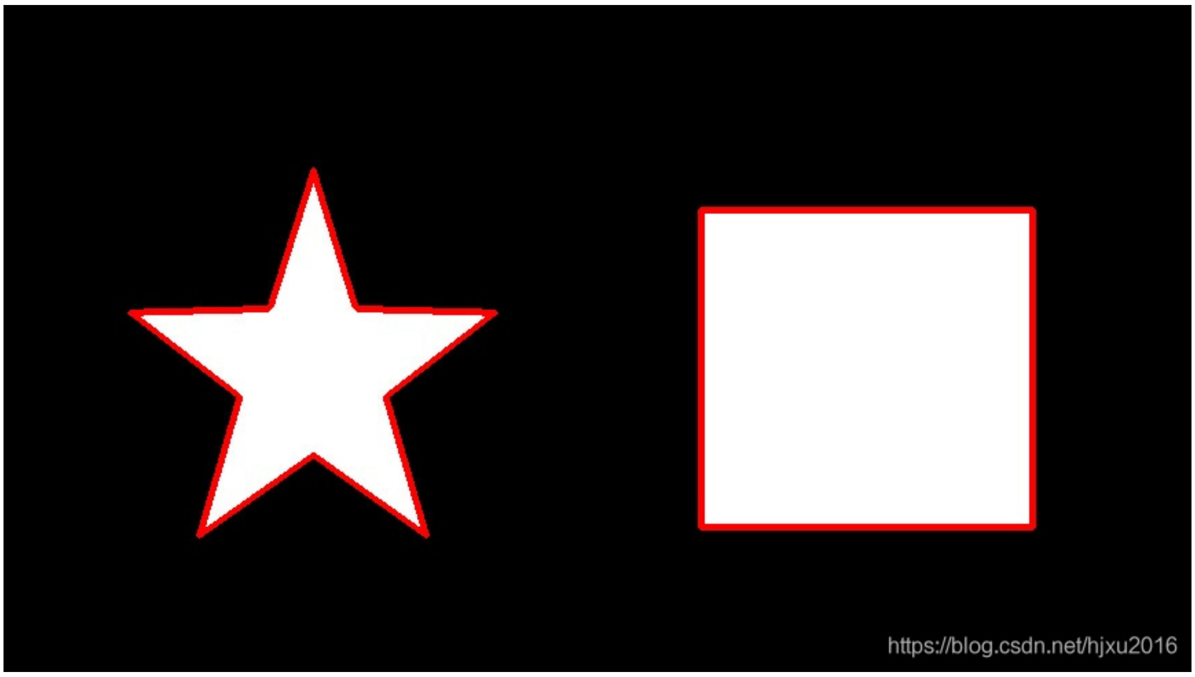
In order to get a better effect, we change our dataset to The Chars74K dataset which has many picture for us to train. And one more thing we do is that we reduce the letter class into 36 classes since some capital letters and small letters are similar and we don't consider that case here.

The job we have done is letter recognition. And we decide to extend our project to word recognition. We use cv2 to incise the word into letters and recognize them one by one.

The general functions we use are cv2.findContours() and cv2.boundingRect():

1.cv2.findContours

contours: Contour points. List format, each element is a 3-dimensional array (of the shape (n,1,2), where n is the number of contour points and 2 is the pixel coordinates), representing a contour



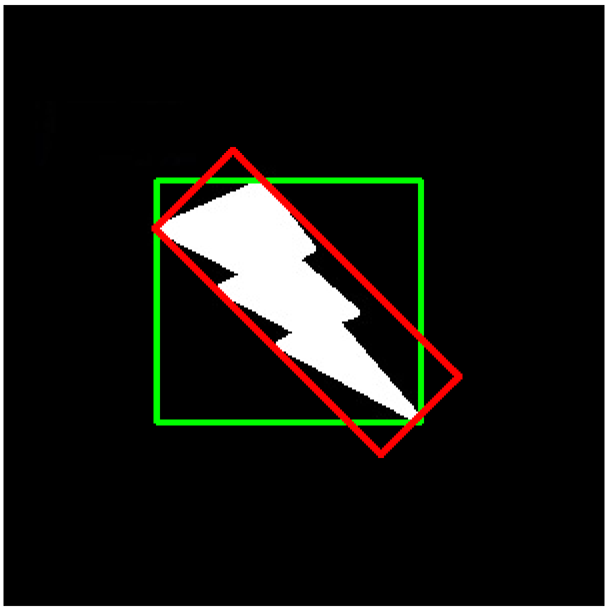
2.cv2.boundingRect():

input:contours

output: x， y， w， h

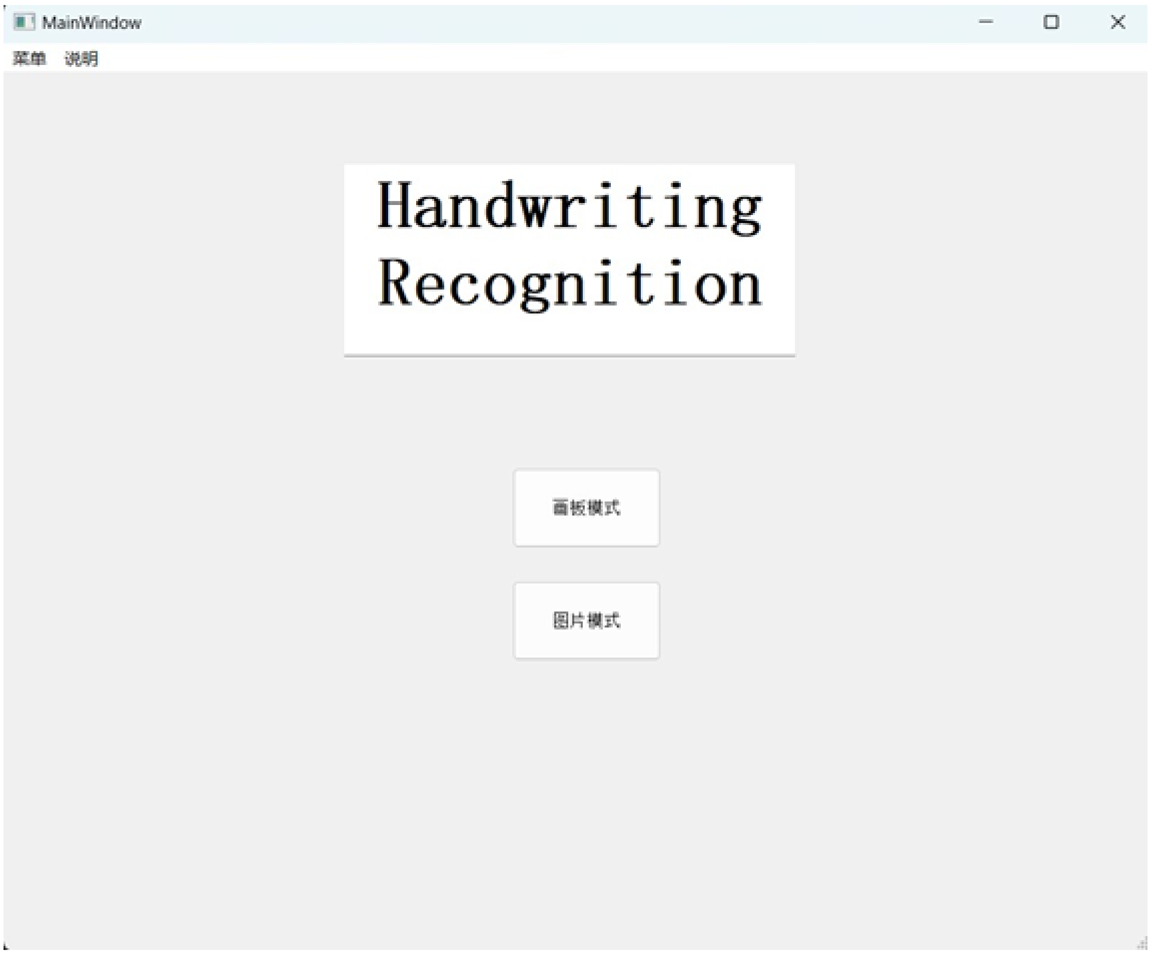
x,y are Coordinates of the upper left point of the matrix.

w,h is the width and height of the rectangular (Green rectangular).

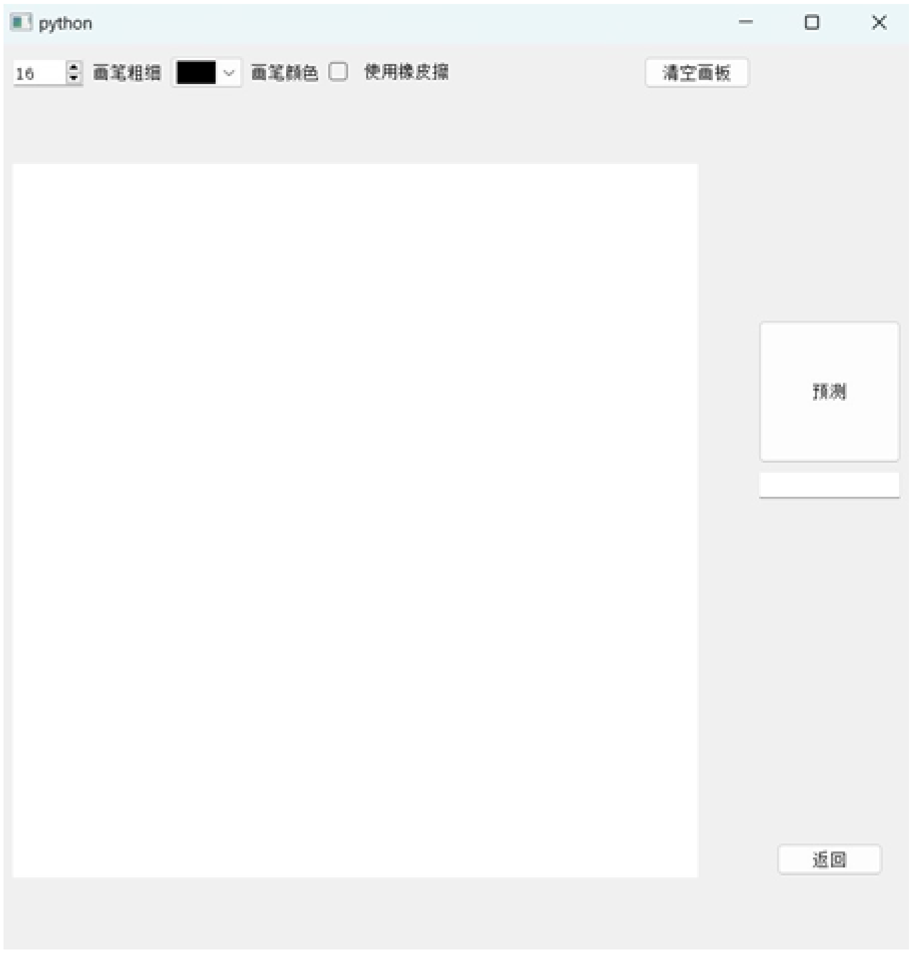


char_image = binary[y:y+h, x:x+w]

we save the binary file and turn it into image to recognize.

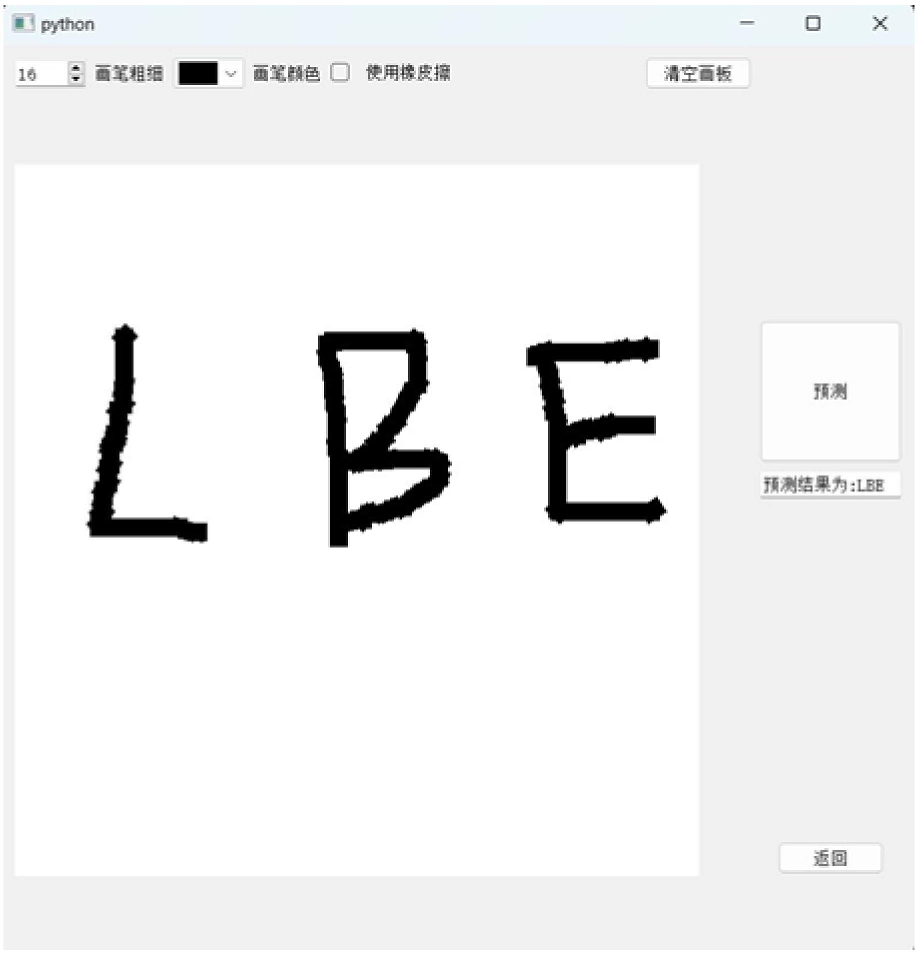


We have two modes of our program. We want to show you the Paint Board Mode first.



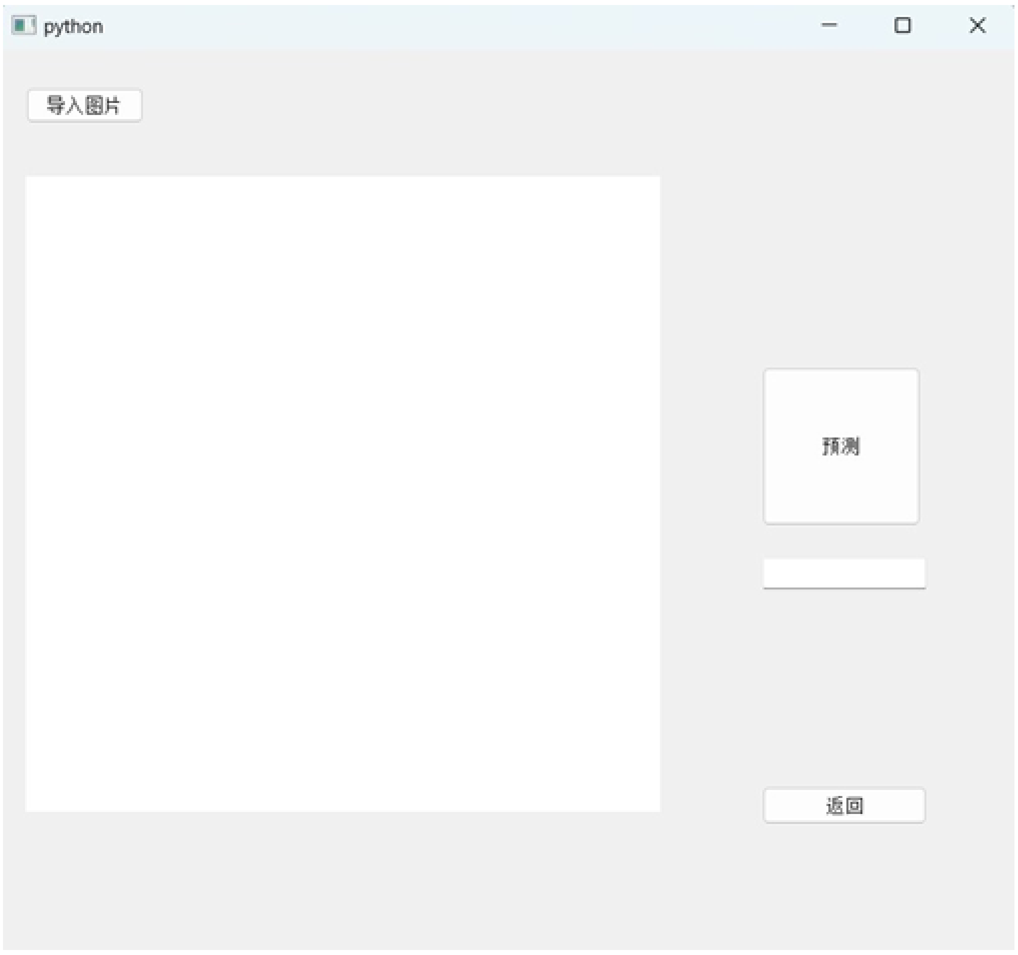
The white part in the left is the paint board. You can use your mouse pointer as a pencil. And you can change the pen tip thickness and color to whatever your want. And if you write something wrong, you can use the eraser to clear.

After you write down the word, you just need to click "预测"，then the recognition result will show in the textline.

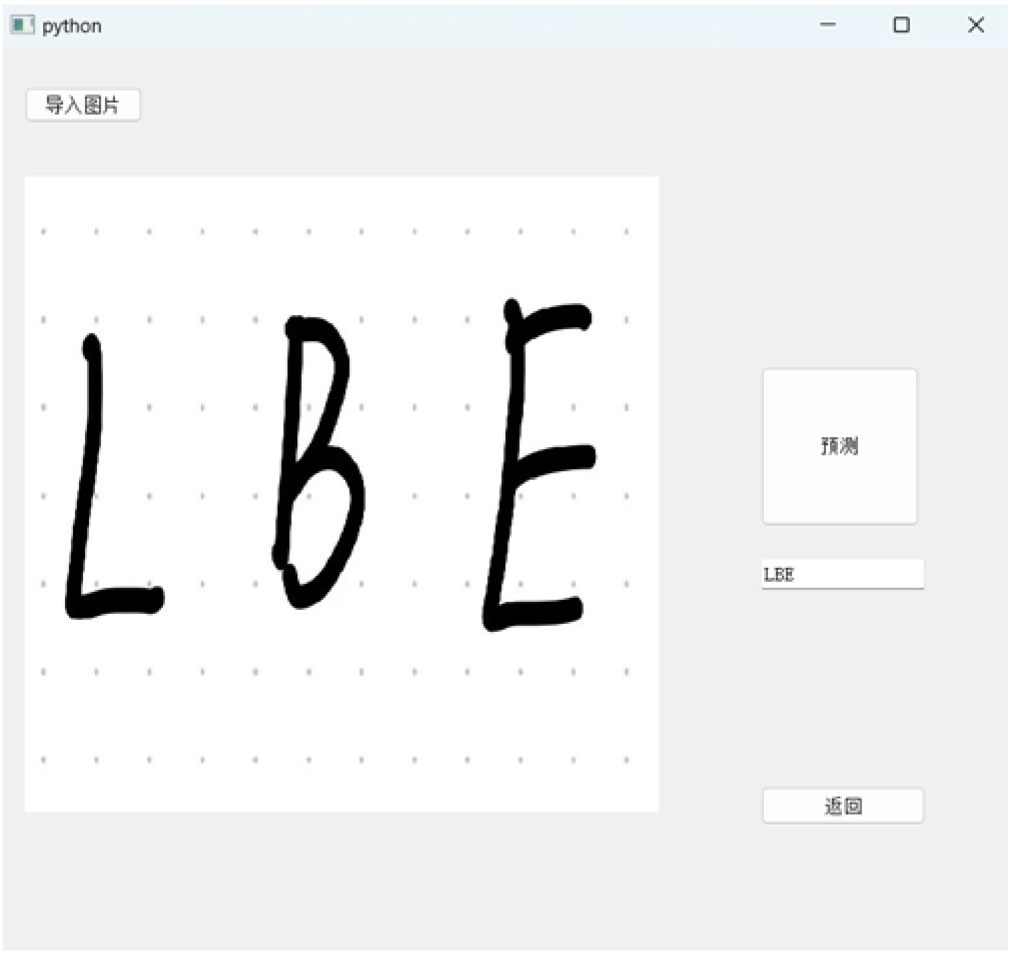


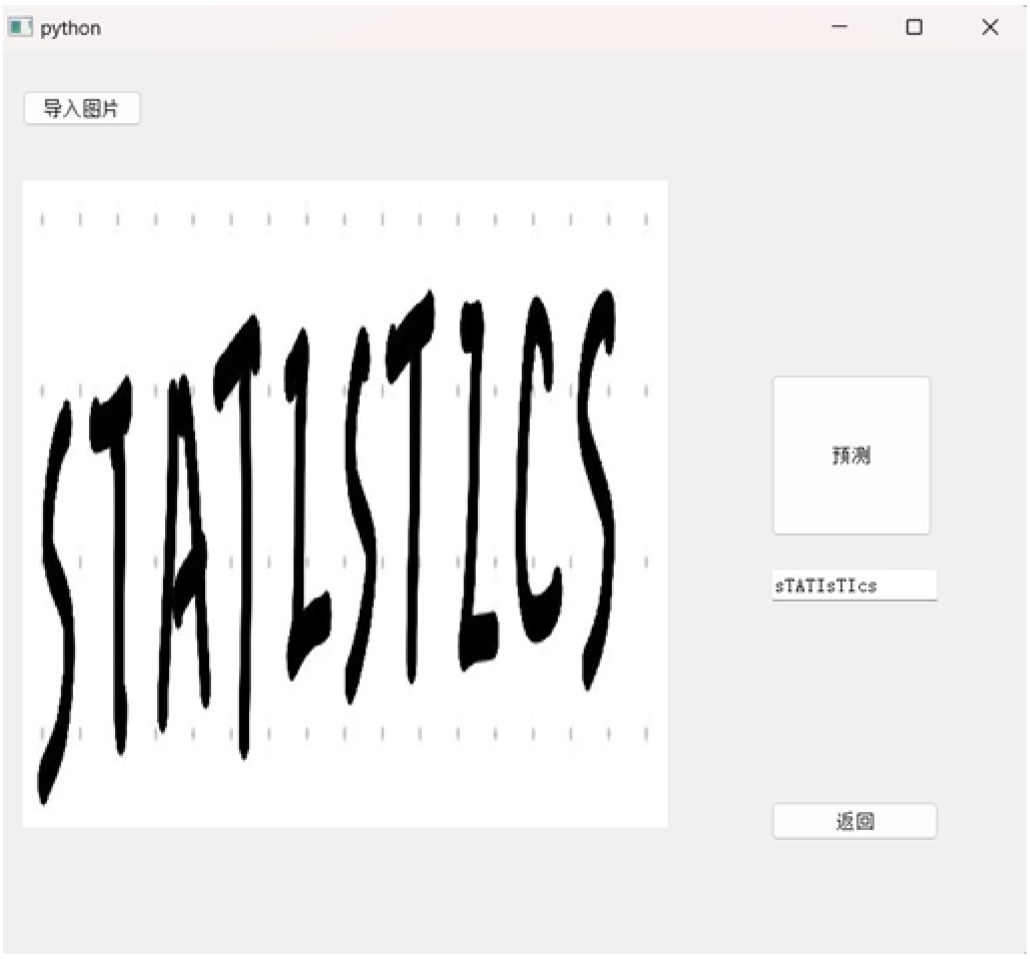
Also, you can click "清空画板", and write down another word. After that, you just need to click "返回", then you can get back to the main window and choose the mode again.

Next, we will show the second mode: Picture Mode.



In this mode, we can use some word we have written or some picture we see. You just need to click "导入图片" to import the picture you are interested.





This seems doing a really good job!

4. Drawbacks

We still have following problems:

- The LeNet is easy to overfitting. We can not reach the accuracy of the training set when testing in practice.
- The accuracy of the AlexNet is lower than 90%
- The photo processing part will incise "i" into two letters because of the dot.

5.Reference

[1] Character Recognition using AlexNet. Minh Thang Dang. 2020. http://dangminhthang.com/computer-vision/character-recognition-using-alexnet/

[2] The Chars74K dataset.http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/

[3] Prashanth, D.S., Mehta, R.V.K., Ramana, K. et al. Handwritten Devanagari Character Recognition Using Modified Lenet and Alexnet Convolution Neural Networks. Wireless Pers Commun 122, 349–378 (2022). https://doi.org/10.1007/s11277-021-08903-4

[4] K. Sadekar, A. Tiwari, P. Singh and S. Raman, "LS-HDIB: A Large Scale Handwritten Document Image Binarization Dataset," 2022 26th International Conference on Pattern Recognition (ICPR), 2022, pp. 1678-1684, doi: 10.1109/ICPR56361.2022.9956447.