Final project: Captcha recognition

Wei-Yun Hsu
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University

1. Introduction

In this final project, we need to train a model to predict all the digits in the image for three tasks. The descriptions of the task are as follows.

- Task 1: Single character in the image.
- Task 2: Two characters in the image (order matters).
- Task 3: Four characters in the image (order matters).

2. Environment details

In this part, we will list our environment details to implement this project. Our environment is created by Anaconda and run all the code in jupyter notebook.

- python: 3.8.13
- pytorch: 1.10.0
  It is mainly used to build each layer for the model and create the dataloader function in this project.
- torchvision: 0.11.0
  It is common for image transformations (or augmenting data) using torchvision.transforms module. They can be chained together using Compose.
- torchsummary: 1.5.1
  Unlike Keras, there is no method in PyTorch nn.Module class to calculate the number of trainable and non-trainable parameters in a model and show the model summary layer-wise. This library can be used to print out the trainable and non-trainable parameters in a Keras-like manner for PyTorch models.
- tqdm: 4.64.1
  It is used to visualize the training progress.
- numpy: 1.23.1
- opencv-python: 4.7.0.72

- Pillow: 9.2.0
  The above three toolboxes are used for reading image and data preprocessing.
- matplotlib: 3.5.2
  The famous matplotlib is used to plot the curve, including accuracy and loss curve in the training phase.

3. Implementation details

A. Model architecture

For the model design, ResNet18 is used for all the experiments in this project. And we implement it using PyTorch from scratch.

ResNet18 is designed to address the problem of vanishing gradients in very deep neural networks. It uses residual blocks, which consists of two or three convolutional layers with a skip connection. This skip connection allows the gradient to flow directly through the block, which can help alleviate the vanishing gradient problem. In the implementation, BasicBlock is used in ResNet18. The key difference between BasicBlock and the other residual block, Bottleneck, is the number of convolutional layers and the size of the filters used within each block, both designs have similar time complexity.

In our implementation, BasicBlock is used, which consists of two convolutional layers with 3x3 filters, followed by a batch normalization layer and a ReLU activation function. This block is repeated multiple times with skip connections between them to form the full network.

The implementation of BasicBlock and ResNet18 are as follows.

```
def downsample(in_ch, out_ch, stride):
    return nn.Sequential(
        nn.Conv2d(in_ch, out_ch, kernel_size=(1, 1), stride=stride, bias=False),
        nn.BatchNorm2d(out_ch))

class BasicBlock(nn.Module):
    '''
    input -> con2d(3x3) -> BN -> activation -> con2d(3x3) -> BN -> activation -> output
    Perform downsampling directly by convolutional layers that have a stride of 2
    '''
    def __init__(self, in_ch, out_ch, downsample_stride):
        super(BasicBlock, self).__init__()
        if downsample_stride is None:
            self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
            self.downsample = None
        else:
            self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
            self.downsample = downsample(in_ch, out_ch, downsample_stride)
        self.bn1 = nn.BatchNorm2d(out_ch)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        self.bn2 = nn.BatchNorm2d(out_ch)

    def forward(self, x):
        ori = x
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.bn2(self.conv2(out))
        if self.downsample is not None:
            ori = self.downsample(ori)
        out = self.relu(out+ori)
        return out
```

```
class ResNet18(nn.Module):
    def __init__(self):
        super(ResNet18, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = nn.Sequential(
            BasicBlock(64, 64, None),
            BasicBlock(64, 64, None)
        )
        self.layer2 = nn.Sequential(
            BasicBlock(64, 128, (2, 2)),
            BasicBlock(128, 128, None)
        )
        self.layer3 = nn.Sequential(
            BasicBlock(128, 256, (2, 2)),
            BasicBlock(256, 256, None)
        )
        self.layer4 = nn.Sequential(
            BasicBlock(256, 512, (2, 2)),
            BasicBlock(512, 512, None)
        )
        self.avgpool = nn.AdaptiveAvgPool2d(output_size=(1, 1))
        self.fc = nn.Linear(512, len(alphabets))
```

```
    def forward(self, x):
        out = self.bn1(self.conv1(x))
        out = self.relu(out)
        out = self.maxpool(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.avgpool(out)
        out = self.fc(out.reshape(out.shape[0], -1))
        return out
```

Additionally, after preparing data, the input data with shape [1, 96, 96] will be fed into model. Therefore, the summary of the model is shown below.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [32, 64, 48, 48]           3,136
       BatchNorm2d-2          [32, 64, 48, 48]             128
              ReLU-3          [32, 64, 48, 48]               0
         MaxPool2d-4          [32, 64, 24, 24]               0
            Conv2d-5          [32, 64, 24, 24]          36,864
       BatchNorm2d-6          [32, 64, 24, 24]             128
              ReLU-7          [32, 64, 24, 24]               0
            Conv2d-8          [32, 64, 24, 24]          36,864
       BatchNorm2d-9          [32, 64, 24, 24]             128
             ReLU-10          [32, 64, 24, 24]               0
       BasicBlock-11          [32, 64, 24, 24]               0
           Conv2d-12          [32, 64, 24, 24]          36,864
      BatchNorm2d-13          [32, 64, 24, 24]             128
             ReLU-14          [32, 64, 24, 24]               0
           Conv2d-15          [32, 64, 24, 24]          36,864
      BatchNorm2d-16          [32, 64, 24, 24]             128
             ReLU-17          [32, 64, 24, 24]               0
       BasicBlock-18          [32, 64, 24, 24]               0
           Conv2d-19         [32, 128, 12, 12]          73,728
      BatchNorm2d-20         [32, 128, 12, 12]             256
             ReLU-21         [32, 128, 12, 12]               0
           Conv2d-22         [32, 128, 12, 12]         147,456
      BatchNorm2d-23         [32, 128, 12, 12]             256
           Conv2d-24         [32, 128, 12, 12]           8,192
      BatchNorm2d-25         [32, 128, 12, 12]             256
             ReLU-26         [32, 128, 12, 12]               0
       BasicBlock-27         [32, 128, 12, 12]               0
           Conv2d-28         [32, 128, 12, 12]         147,456
      BatchNorm2d-29         [32, 128, 12, 12]             256
             ReLU-30         [32, 128, 12, 12]               0
                                    .
                                    .
                                    .
                                    .
                                    .
                                    .
      BatchNorm2d-64           [32, 512, 3, 3]           1,024
             ReLU-65           [32, 512, 3, 3]               0
       BasicBlock-66           [32, 512, 3, 3]               0
AdaptiveAvgPool2d-67           [32, 512, 1, 1]               0
           Linear-68                 [32, 62]          31,806
================================================================
Total params: 11,202,046
Trainable params: 11,202,046
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 1.12
Forward/backward pass size (MB): 369.14
Params size (MB): 42.73
Estimated Total Size (MB): 413.00
----------------------------------------------------------------
```

B. Experiment setups

The hyper-parameters for experiments as shown below.

Task 1

- epoch: 100
- batch size: 32
- learning rate: 1e-03
- weight decay: 5e-03
- optimizer: Adam

scheduler

- step_size: 10
- gamma: 0.9

Task 2

- epoch: 250
- batch size: 32
- learning rate: 1e-02
- weight decay: 5e-03
- optimizer: Adam

scheduler

- step_size: 30
- gamma: 0.9

Task 3

- epoch: 200
- batch size: 32
- learning rate: 1e-04
- weight decay: 5e-03
- optimizer: Adam

scheduler

- step_size: 20
- gamma: 0.5

C. Experimental design

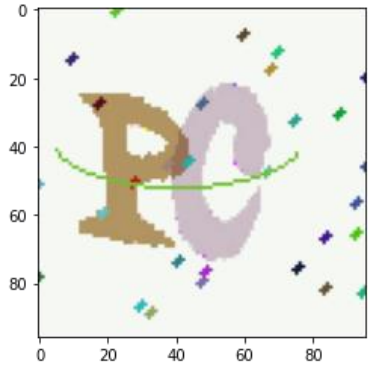
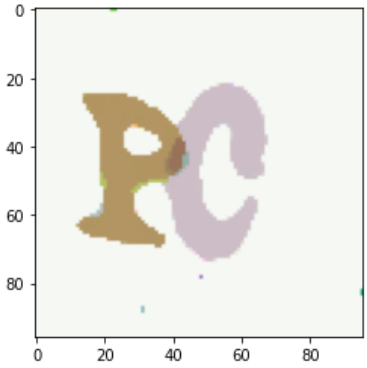
In this section, we will describe our entire experimental process in detail, including data preprocessing, data augmentation, and the training loop.

C-1. Data preprocessing

At the very beginning, we thought that the noise point in the background would not have much impact on recognizing the digits in the image, so we only used rotation to the image and retained the size of the original image. But in the initial experimental results, we found that the accuracy seemed to be a bottleneck. For this reason, we browsed the source image and observed those noise points may increase the difficulty of recognition. Therefore, we slightly modify the data preprocessing step.

The filter is a good technique, and the median filter is the most suitable in our case. The median filter is a non-linear digital filtering technique, often used to remove noise from an image or signal. It runs through each element of the image and replaces each pixel with the median of its neighboring pixels which located in a square neighborhood around the evaluated pixel. To achieve the above result, we

need to use directly the medianBlur() function of the OpenCV. After pass through the median filter, we convert the RGB image to grayscale image by calculating the mean value along the color channel axis, the resulting image will contain only one channel representing the overall intensity or brightness of the original image. The effect of the median filter is shown below.

Before filtering	After filtering
	

In addition, in order to generalize our model, we also applied the rotation in our data augmentation method

The implementations of the dataloader for three tasks are as follows.

```
class Task1Dataset(Dataset):
    def __init__(self, data, root, return_filename=False):
        self.data = [sample for sample in data if sample[0].startswith("task1")]
        self.return_filename = return_filename
        self.root = root
        self.transform = transforms.Compose([
            #transforms.Resize(32),
            transforms.RandomRotation(degrees=20),
            transforms.ToTensor()
        ])

    def __getitem__(self, index):
        filename, label = self.data[index]
        img = cv2.imread("{}/{}".format(self.root, filename))
        img = cv2.resize(img, (96, 96))
        img = cv2.medianBlur(img, 5)
        img = np.mean(img, axis=2)
        img = Image.fromarray(img)
        img = self.transform(img)

        if self.return_filename:
            return torch.FloatTensor((img - 128) / 128), filename
        else:
            return torch.FloatTensor((img - 128) / 128), alphabets2index[label]

    def __len__(self):
        return len(self.data)
```

```
class Task2Dataset(Dataset):
    def __init__(self, data, root, return_filename=False):
        self.data = [sample for sample in data if sample[0].startswith("task2")]
        self.return_filename = return_filename
        self.root = root
        self.transform = transforms.Compose([
            #transforms.Resize(32),
            transforms.RandomRotation(degrees=20),
            transforms.ToTensor()
        ])

    def __getitem__(self, index):
        filename, label = self.data[index]
        img = cv2.imread("{}/{}".format(self.root, filename))
        img = cv2.resize(img, (96, 96))
        img = cv2.medianBlur(img, 5)
        img = np.mean(img, axis=2)
        img = Image.fromarray(img)
        img = self.transform(img)
```

```
        label_list = [[alphabets2index[digit]] for digit in label]
        label_list = np.array(label_list)
        #print(label_list)

        length = [len(label_list)]

        #label_list = [length] + label_list
        label_list = np.append([length], label_list, axis=0)

        if self.return_filename:
            return torch.FloatTensor((img - 128) / 128), filename
        else:
            return torch.FloatTensor((img - 128) / 128), label_list

    def __len__(self):
        return len(self.data)
```

```
class Task3Dataset(Dataset):
    def __init__(self, data, root, return_filename=False):
        self.data = [sample for sample in data if sample[0].startswith("task3")]
        self.return_filename = return_filename
        self.root = root

        self.transform = transforms.Compose([
            #transforms.Resize(96),
            transforms.RandomRotation(degrees=20),
            transforms.ToTensor()
        ])

    def __getitem__(self, index):
        filename, label = self.data[index]
        img = cv2.imread("{}/{}".format(self.root, filename))
        img = cv2.resize(img, (96, 96))
        img = cv2.medianBlur(img, 5)
        img = np.mean(img, axis=2)
        img = Image.fromarray(img)
        img = self.transform(img)

        label_list = [[alphabets2index[digit]] for digit in label]
        label_list = np.array(label_list)
        #print(label_list)

        length = [len(label_list)]
```

```
        #label_list = [length] + label_list
        label_list = np.append([length], label_list, axis=0)

        if self.return_filename:
            return torch.FloatTensor((img - 128) / 128), filename
        else:
            return torch.FloatTensor((img - 128) / 128), label_list

    def __len__(self):
        return len(self.data)
```

C-2. Methodology

As described in section A, although we used ResNet18 to recognize all the tasks, the number of character for each task is different, so we had a specific ResNet18 architecture for each task. You can refer to `311553009_weight.txt` and load the model weights for three tasks respectively.

The output design of ResNet18 for each task, nn.Linear() is applied. The model returns not only the index of its predicted number but also the length, that is, the number of characters in the image, and then the model will compute the loss for them individually. Finally, we get the loss predicted by the current data using adding them together.

Taking the task3 for an example, the following figures are the output layer and the loss summation in the training phase.

```
    self.digit1 = nn.Linear(512, len(alphabets))
    self.digit2 = nn.Linear(512, len(alphabets))
    self.digit3 = nn.Linear(512, len(alphabets))
    self.digit4 = nn.Linear(512, len(alphabets))
```

Figure 1. The output layer in the __init__() function

```
  yl = self.digitlength(out.reshape(out.shape[0], -1))
  y1 = self.digit1(out.reshape(out.shape[0], -1))
  y2 = self.digit2(out.reshape(out.shape[0], -1))
  y3 = self.digit3(out.reshape(out.shape[0], -1))
  y4 = self.digit4(out.reshape(out.shape[0], -1))
  return [yl, y1, y2, y3, y4]
```

Figure 2. The output layer in the forward() function

```
lossl = loss_fn(outputs[0][idxs], Y[:, 0])
loss1 = loss_fn(outputs[1][idxs], torch.tensor(Y[:, 1].squeeze(1), dtype=torch.int64))
loss2 = loss_fn(outputs[2][idxs], torch.tensor(Y[:, 2].squeeze(1), dtype=torch.int64))
loss3 = loss_fn(outputs[3][idxs], torch.tensor(Y[:, 3].squeeze(1), dtype=torch.int64))
loss4 = loss_fn(outputs[4][idxs], torch.tensor(Y[:, 4].squeeze(1), dtype=torch.int64))
lossd2 = lossl + loss1 + loss2 + loss3 + loss4
```

Figure 3. The loss computation in our training phase

On the other hand, a special thing is the reward design when the model predict multiple digits. We tried to choose the best mechanism, which can make the model develop toward the better direction. Taking task3 as an example again, the model will get some rewards when one of the digits in the image is correctly predicted instead of predicting all four numbers correctly. This mechanism can help the model to learn the importance of feature extraction. The implementation is as

follows.

```
for i in range(idx+1):
    if (pred1[i] == torch.tensor(Y[:, 1].squeeze(1), dtype=torch.int64)[i]):
        correct_count += 0.25
    if (pred2[i] == torch.tensor(Y[:, 2].squeeze(1), dtype=torch.int64)[i]):
        correct_count += 0.25
    if (pred3[i] == torch.tensor(Y[:, 3].squeeze(1), dtype=torch.int64)[i]):
        correct_count += 0.25
    if (pred4[i] == torch.tensor(Y[:, 4].squeeze(1), dtype=torch.int64)[i]):
        correct_count += 0.25
```

You can refer to `311553009_train.ipynb` for more details.

D. Results and discussion

D-1. Results

In this part, we will show the training curve and the accuracy for training and validation, respectively.

- **Task 1**

	Training acc	Training loss	Validation acc	Validation loss
without noise removal	0.95	0.0387	0.8125	0.5848
with noise removal	1	0.0211	0.967	0.1614

Table 1. The statistics for task1

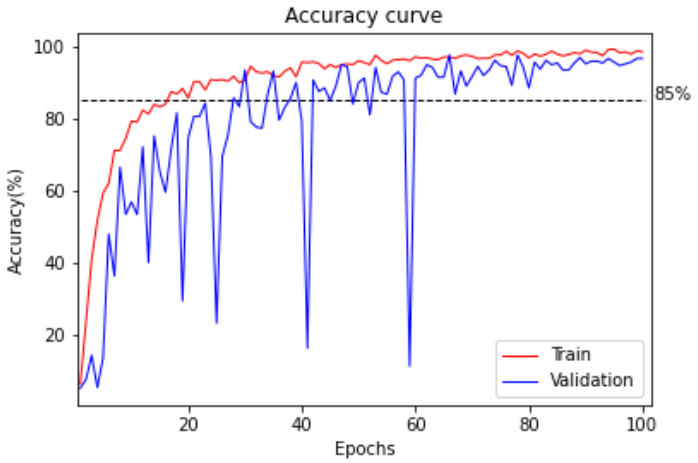


Figure 4. The accuracy curve for task1

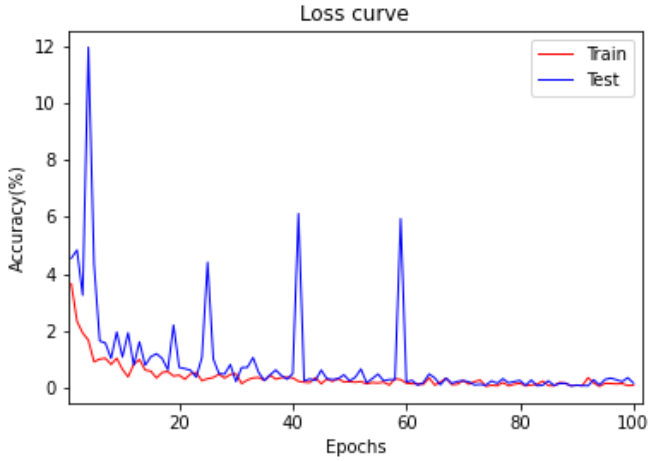


Figure 5. The loss curve for task1

- **Task 2**

	Training acc	Training loss	Validation acc	Validation loss
without noise removal	0.9364	0.233	0.3413	4.7677
with noise removal	0.9558	0.817	0.8142	1.5158

Table 2. The statistics for task2

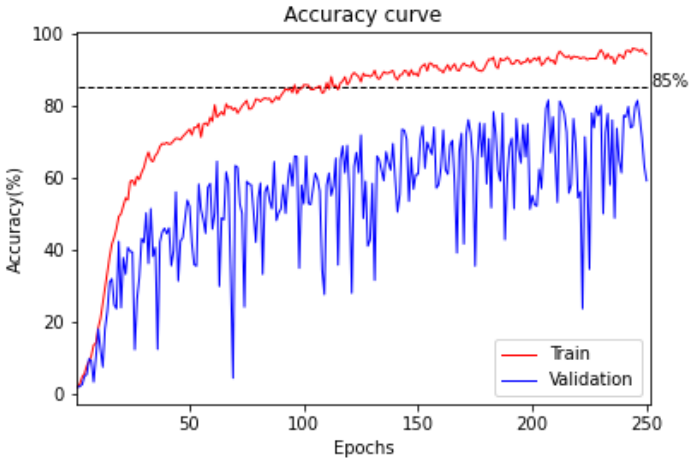


Figure 6. The accuracy curve for task2

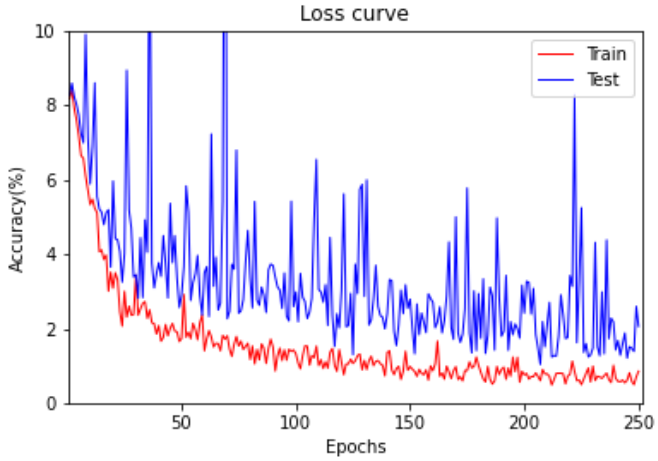


Figure 7. The loss curve for task2

- **Task 3**

	Training acc	Training loss	Validation acc	Validation loss
without noise removal	0.9892	0.2328	0.311	13.8044
with noise removal	0.9947	0.2251	0.5046	10.7572

Table 3. The statistics for task3

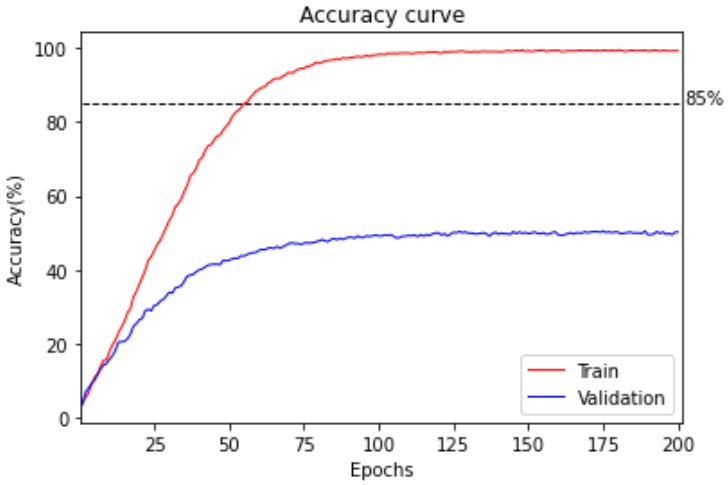


Figure 8. The accuracy curve for task3

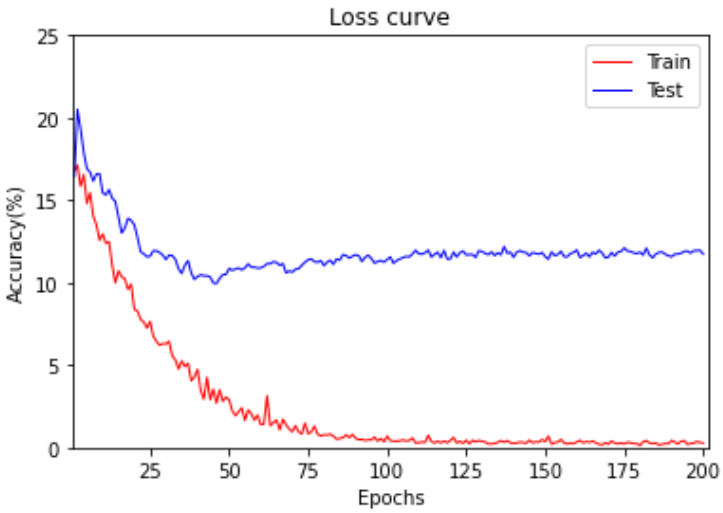


Figure 9. The loss curve for task3

D-2. Discussion

- **Different settings of learning rate**

  Before doing this project, we planned to use the learning rate decay strategy to train our model, because without using learning rate decay, the optimizer can overshoot the optimal solution and cause the loss function to diverge, the model will fail to learn. We tried to start with a relatively high learning rate and gradually reduced it as training progresses. This allows the optimizer to take large steps in the beginning when the parameters are far from the optimal solutions, but reduces the step size as the parameters get closer to the optimal solutions to help the optimizer converge.

  During the experiment, we observed that it is difficult to choose a good initial learning rate for task3, since it is the more complex problem compare with the other tasks. The experimental results violated our original thinking, which is the more complex problem need the larger learning rate to train our model. In fact, for task3, whether the learning rate is too large or too small, the model cannot learn toward the expected direction, it may not converge or even diverge.

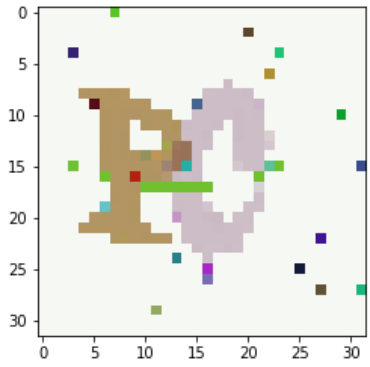
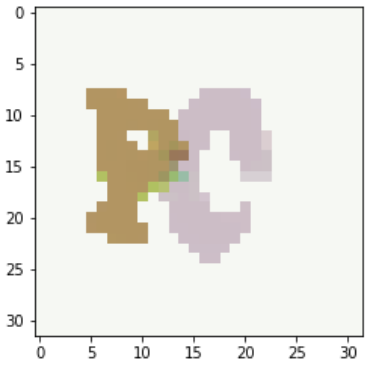
- **Trade-off between the image shape and the window size of median filter**

  From the section 3-C, we can see that we will get the smoother image after filtering by background noise reduction. However, there is a trade-off problem, that is, how to achieve a balance between the image shape and the window size of the filter. Assume that we have an image with shape [32, 32], the smaller filter size is needed without overly blurring objects in the image, but the effect of denoising isn't obvious. On the contrary, if we have an image

with shape [96, 96], then we expected the larger filter size to apply on the image, but it may cause the overly smoothing problem. The comparison results are shown below.

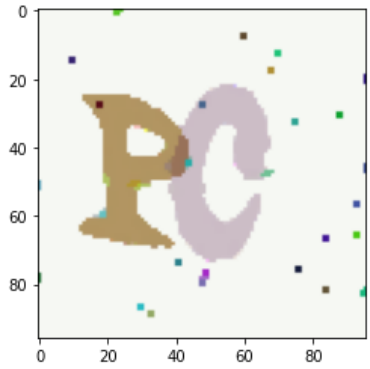
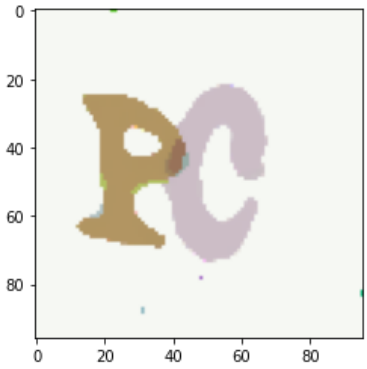
- Image shape: [32, 32]

wz: window size

Filtering with the smaller wz	Filtering with the larger wz
	

- Image shape: [96, 96]

wz: window size

Filtering with the smaller wz	Filtering with the larger wz
	

From the above reasons, we decided to retain the shape of the original image and applied a median filter with the larger window size as our experimental settings.