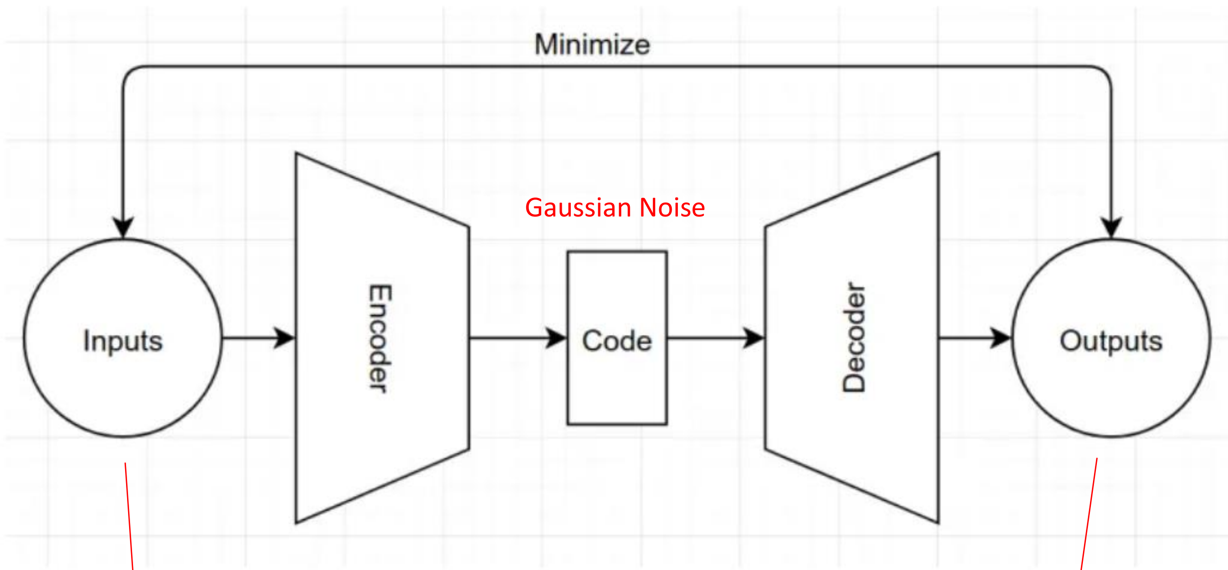
HW2 Report_ Convolutional Autoencoder_109064518_高聖哲

1. Model Arichitecture



-data_npy 格式 :(1281,26,26,3) → gen_data_npy 格式 :(105,61,26,26,3)

-label_npy 格式:(1281,1) → gen_label_npy 格式 :(105,61,1)

I. Model 敘述:

a. 架構中可細分為 Encoder（編碼器）和 Decoder（解碼器）兩部分，它們分別做壓縮與解壓縮的動作，讓輸出值和輸入值表示相同.

b. 在 AutoEncoder 過程中增加了一些限制，使生成向量遵從高斯分佈.由於高斯分佈可以通過其 mean 和 standard deviation 進行參數化，因此是可以讓我們控制要生成的圖片。

II. Model building block

input_size(int) –輸入訊號的通道數
output_size(int) –卷積產生的通道數
kerner_size(int or tuple) – 卷積核的大小
stride(int or tuple,optional) – 卷積步長，即要輸入擴大的倍數
padding(int or tuple, optional) –輸入的每一條邊補充 0 的層數,高寬都增加 2*padding
outpadding(int or tuple, optional) – 輸出邊補充 0 的層數，高寬都增加 padding

a. **Encoder**

i. Conv2D 公式：(inputsize-kernel+2*padding)/stride+1

-conv layer:(depth from 3 --> 16), 3x3 kernels

```
self.conv1 = nn.Conv2d(3, 16, 3, stride=2, padding=1)
```

-conv layer:(depth from 16 --> 32), 3x3 kernels

```
self.conv2 = nn.Conv2d(16, 32, 3, stride=2, padding=1)
```

-conv layer:(depth from 32 --> 64), 5x5 kernels

```
self.conv3 = nn.Conv2d(32, 64, 5)
```

ii. 透過 Relu 的 activation function 來新增 hidden layer

```
## encode ##
  # add hidden layers with relu activation function
  # add first hidden layer
  x = F.relu(self.conv1(x))

  # add second hidden layer
  x = F.relu(self.conv2(x))

  x = self.conv3(x)
```

b. Decoder

i. ConvTranspose2d 公式:(inputsize-1)*stride+kernel-2*padding+outpadding

-tconv layer (depth from 64 --> 32), 5x5 kernels

```
self.t_conv1 = nn.ConvTranspose2d(64, 32, 5)
```

-tconv layer (depth from 32 --> 16), 3x3 kernels

```
self.t_conv2 = nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1)
```

-tconv layer (depth from 16 --> 3), 3x3 kernels

```
self.t_conv3 = nn.ConvTranspose2d(16, 3, 3, stride=2, padding=2, output_padding=1)
```

ii. 透過 Relu 的 activation function 來新增 transpose con layer

```
## decode ##
# add transpose conv layers, with relu activation function
x = F.relu(self.t_conv1(x))
x = F.relu(self.t_conv2(x))
```

iii. 使用 sigmoid 作為 output layer,最後的 x 輸出剛好等於 input 的 size,則可以做後續的 loss 計算

```
# output layer (with sigmoid for scaling from 0 to 1)
x = F.sigmoid(self.t_conv3(x))
```

c. Loss function

i. nn.MSELoss 均方損失函数： $loss(x_i, y_i) = (x_i - y_i)^2$,這裡的 loss, x, y 的維度是一樣的,可以是向量或是矩陣，i 是下標.比如若 x, y, 是矩陣 $x = [a_{ij}], y = [b_{ij}], 0 < i < n, 0 < j < m$

ii. Adam 演算法：利用梯度的一階矩估計和二階矩估計動態調整每個引數的學習

```
# specify loss function
criterion = nn.MSELoss()
# specify loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.001,weight_decay=1e-5)
```

iii. Output=model(new_image1):對應前向傳播求出預測值

iv. loss = criterion(outputs, new_image1):求當前的 loss 值

v. optimizer.zero_grad():梯度置零，也就是把 loss 關於 weight 的導數變成 0.

vi. loss.backward():對應反向傳播求梯度

vii. optimizer.step():對應更新所有參數

```
# forward pass: compute predicted outputs by passing inputs to the model
outputs = model(image_test)#output is reconstruction image

# calculate the loss
loss = criterion(outputs, image_test)#calculate the reconstruction image and original image
# clear the gradients of all optimized variables
optimizer.zero_grad()
# backward pass: compute gradient of the loss with respect to model parameters
loss.backward()
# perform a single optimization step (parameter update)
optimizer.step()
```

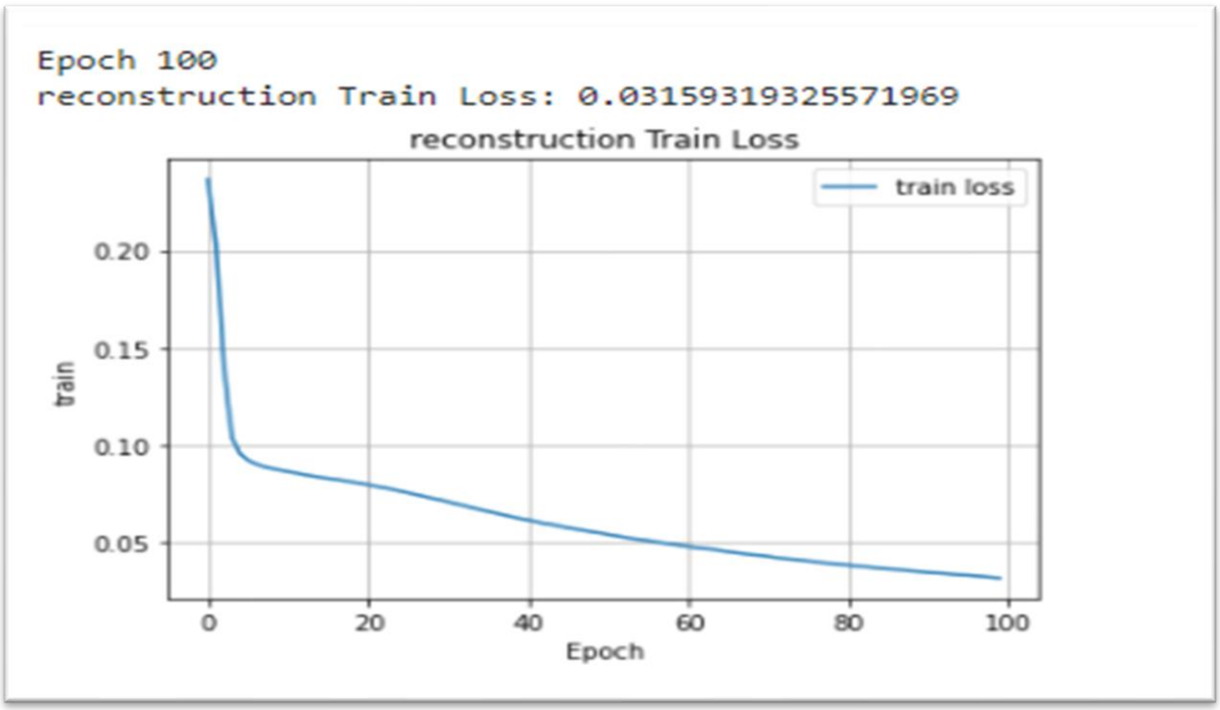
d. Gaussian noise

i. torch.randn_like()函式:創建和 input 同樣尺寸的 noisy tensor 來達到 Gaussian 的效果,並加 add_noise function 加到 autoencoder 的 latent code 中

```
def add_noise(inputs,i):
        noise = torch.randn_like(inputs)*(i/10)
        return inputs + noise
```

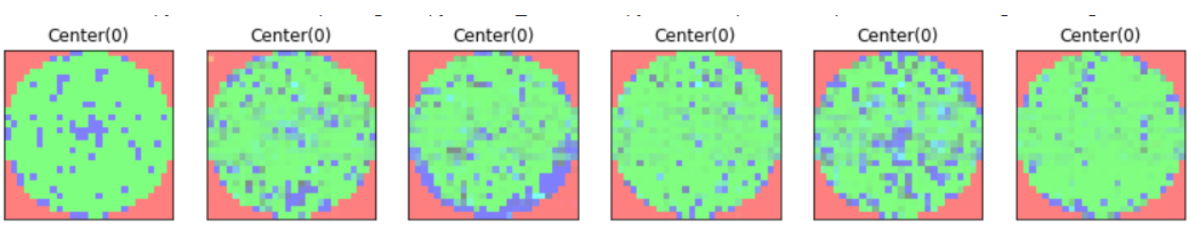
III. Model 模型結果

a. Learning rate:0.001
b. Weight-decay:1e-5
c. Epoach:100
d. Batchsize:61
e. Reconstruction training loss: 0.03159319325571969
f. Training figure:

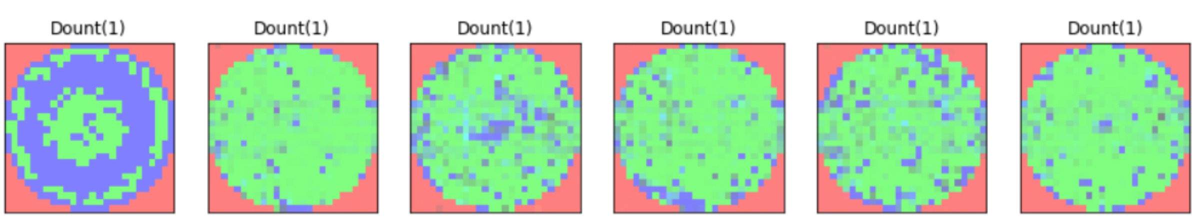


IV. Visualize 5 generated samples for each class

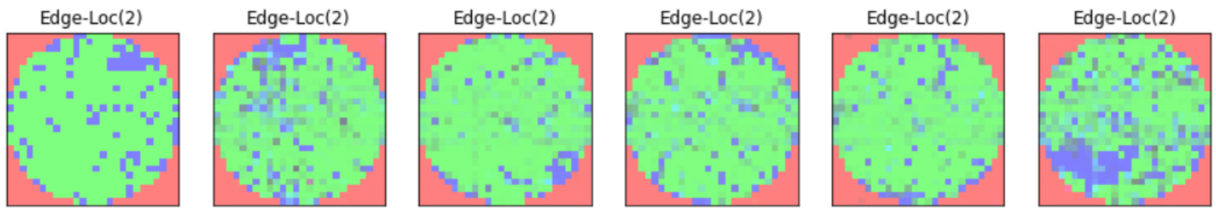
a. Center(0)



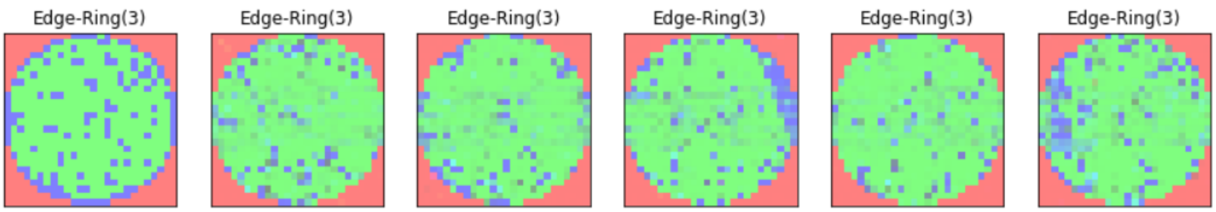
b. Dount(1)



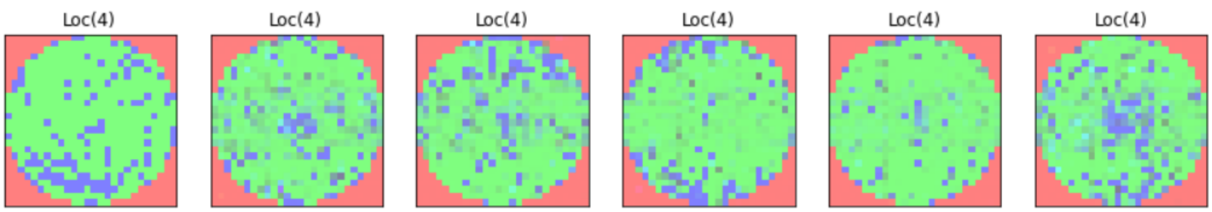
c. Edge-Loc(2)



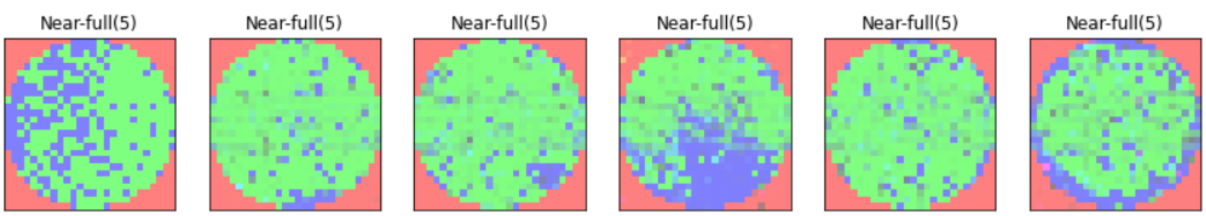
d. Edge-Ring(3)



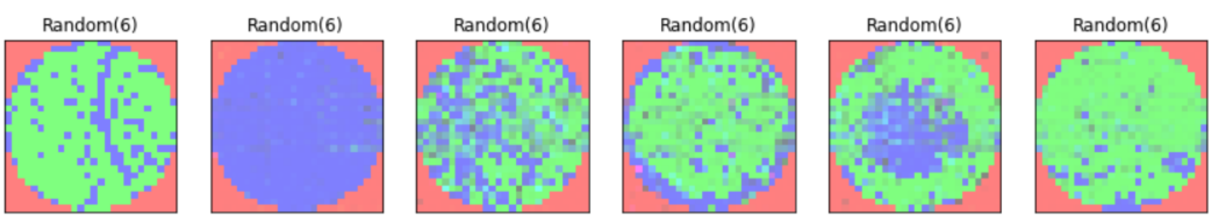
e. Loc(4)



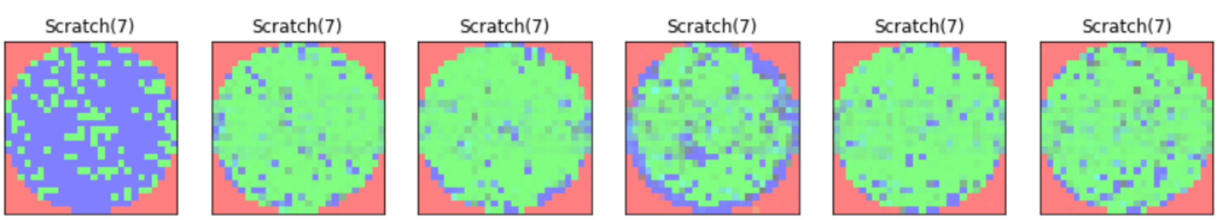
f. Near-full(5)



g. Random(6)



h. Scratch(7)



i. None(8)