Problem 3

As usual, import necessary modules, and variables.

In [ ]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from torchvision import datasets, transforms
from torchvision.utils import save_image, make_grid

import numpy as np
import matplotlib.pyplot as plt

import warnings
warnings.filterwarnings('ignore')

batch_size = 128
(full_dim, mid_dim, hidden) = (1 * 28 * 28, 1000, 5)
lr = 1e-3
epochs = 100
device = torch.device("cpu")
```

In [ ]:

```
class Logistic(torch.distributions.Distribution):
    def __init__(self):
        super(Logistic, self).__init__()

    def log_prob(self, x):
        return -(F.softplus(x) + F.softplus(-x))

    def sample(self, size):
        z = torch.distributions.Uniform(0., 1.).sample(size).to(device)
        return torch.log(z) - torch.log(1. - z)

class Coupling(nn.Module):
    def __init__(self, in_out_dim, mid_dim, hidden, mask_config):
        super(Coupling, self).__init__()
        self.mask_config = mask_config

        self.in_block = nn.Sequential(nn.Linear(in_out_dim//2, mid_dim), nn.ReLU())
        self.mid_block = nn.ModuleList([nn.Sequential(nn.Linear(mid_dim, mid_dim), 
                                                                  for _ in range(hid
        self.out_block = nn.Linear(mid_dim, in_out_dim//2)

    def forward(self, x, reverse=False):
        [B, W] = list(x.size())
        x = x.reshape((B, W//2, 2))
        if self.mask_config:
            on, off = x[:, :, 0], x[:, :, 1]
        else:
            off, on = x[:, :, 0], x[:, :, 1]

        off_ = self.in_block(off)
        for i in range(len(self.mid_block)):
            off_ = self.mid_block[i](off_)
        shift = self.out_block(off_)

        if reverse:

        else:
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
            on = on + shift

        if self.mask_config:
            x = torch.stack((on, off), dim=2)
        else:
            x = torch.stack((off, on), dim=2)
        return x.reshape((B, W))

class Scaling(nn.Module):
    def __init__(self, dim):
        super(Scaling, self).__init__()
        self.scale = nn.Parameter(torch.zeros((1, dim)), requires_grad=True)

    def forward(self, x, reverse=False):
        log_det_J = torch.sum(self.scale)
        if reverse:
            x = x * torch.exp(-self.scale)
        else:
            x = x * torch.exp(self.scale)
        return x, log_det_J

class NICE(nn.Module):
    def __init__(self,in_out_dim, mid_dim, hidden, mask_config=1.0, coupling=4):
        super(NICE, self).__init__()
        self.prior = Logistic()
        self.in_out_dim = in_out_dim

        self.coupling = nn.ModuleList([
            Coupling(in_out_dim=in_out_dim,
                     mid_dim=mid_dim,
                     hidden=hidden,
                     mask_config=(mask_config+i)%2) ₩
            for i in range(coupling)])

        self.scaling = Scaling(in_out_dim)

    def g(self, z):
        x, _ = self.scaling(z, reverse=True)
        for i in reversed(range(len(self.coupling))):
            x = self.coupling[i](x, reverse=True)
        return x

    def f(self, x):
        for i in range(len(self.coupling)):
            x = self.coupling[i](x)
        z, log_det_J = self.scaling(x)
        return z, log_det_J

    def log_prob(self, x):
        z, log_det_J = self.f(x)
        log_ll = torch.sum(self.prior.log_prob(z), dim=1)
        return log_ll + log_det_J

    def sample(self, size):
        z = self.prior.sample((size, self.in_out_dim)).to(device)
        return self.g(z)

    def forward(self, x):
        return self.log_prob(x)
```

In [ ]:

```
model = NICE(in_out_dim=784, mid_dim=1000, hidden=5).to(device)
ce.pt',map_location=torch.device('cpu')))
```

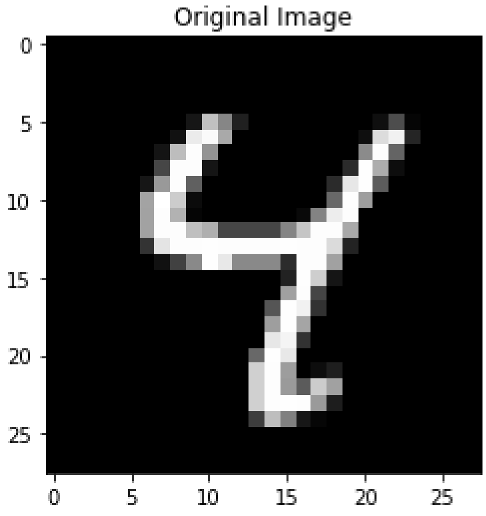
Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
model.requires_grad_(False)
```

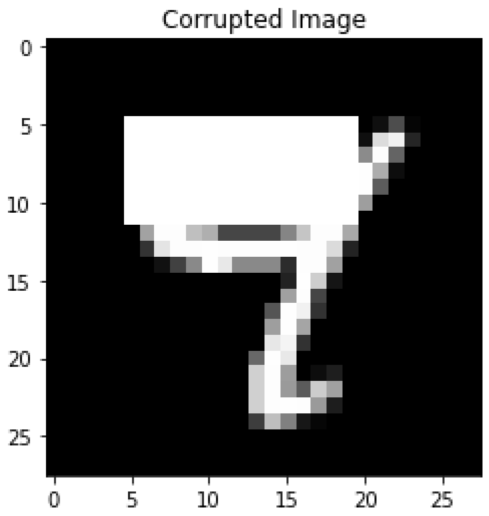
Load dataset and define mask.

In [ ]:

```
testset = torchvision.datasets.MNIST(root='./', train=False, download=True, transf
test_loader = torch.utils.data.DataLoader(testset, batch_size=1, shuffle=False)
pass_count = 6
itr = iter(test_loader)
for _ in range(pass_count+1):
    image,_ = itr.next()

plt.figure(figsize = (4,4))
plt.title('Original Image')
plt.imshow(make_grid(image.squeeze().detach()).permute(1,2,0))
plt.show()
# plt.savefig('plt1.png')


# Create mask
mask = torch.ones_like(image,dtype=torch.bool)
mask[:,:,5:12,5:20] = 0

# Partially corrupt the image
image[mask.logical_not()] = torch.ones_like(image[mask.logical_not()])
plt.figure(figsize = (4,4))
plt.title('Corrupted Image')
plt.imshow(make_grid(image.squeeze()).permute(1,2,0))
plt.show()
# plt.savefig('plt2.png')
```



Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js



Now, we define custom inpainting of single image.
We start from corrupted image, and modifying corrupted part only (by constraints) to maximize likelihood (from NICE).
To do this, we define new module "inpaint", and set its parameter as image pixels.
Since we are using projective gradient, which includes projection part after usual gradient descent, we add projecting part in our module.

In [ ]:

```
class inpaint(nn.Module):
    def __init__(self, raw_image : torch.Tensor, mask : torch.Tensor):
        '''
        image : fix size (1,1,28,28) due to the mask
        '''
        super().__init__()
        self.mask = mask
        self.image = raw_image.clone().requires_grad_(False)
        self.corrupted_part = nn.Parameter(raw_image.clone(), requires_grad=True)

    def forward(self, x):     # x always be a mask
        image = x*self.image + x.logical_not()*self.corrupted_part
        image = image.reshape((1,28*28))
        return model(image)

    def project_gradient(self):
        weight = self.corrupted_part.data
        weight = torch.clamp(self.mask*self.image + self.mask.logical_not()*weight,
        self.corrupted_part.data = weight

inpaint_model = inpaint(image, mask)
inpaint_optimizer = torch.optim.SGD(params = inpaint_model.parameters(), lr = lr)

for i in range(300):
    inpaint_optimizer.zero_grad()
    loss = -inpaint_model(mask)
    loss.backward()
    inpaint_optimizer.step()
    inpaint_model.project_gradient()
```

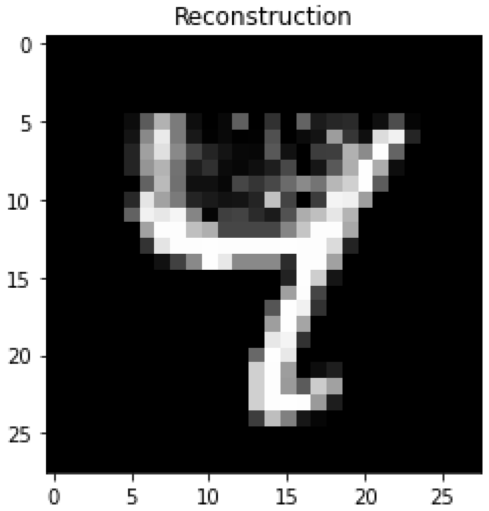
The result looks like following.

In [ ]:

```
recon = inpaint_model.corrupted_part.data
plt.figure(figsize = (4,4))
plt.title('Reconstruction')
                                        .detach()).permute(1,2,0))
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

```
plt.show()
# plt.savefig('plt3.png')
```


Reconstruction