#1. Log-derivative trick for VAE.

$z \in \mathbb{R}^k$: r.v, $q_\phi(z)$: pdf for $\phi \in \mathbb{R}^l$, diff. $h: \mathbb{R}^k \to \mathbb{R}$, $h(z) \geq 0$ ($\forall z \in \mathbb{R}^k$)

$$\nabla_\phi \mathbb{E}_{z \sim q_\phi(z)}\left[\log\left(\frac{h(z)}{q_\phi(z)}\right)\right]$$

$$= \nabla_\phi \int q_\phi(z) \log \frac{h(z)}{q_\phi(z)}\, dz = \int \nabla_\phi \Big(q_\phi(z) \log h(z) - q_\phi(z) \log q_\phi(z)\Big)\, dz$$

$$= \int \Big(\log h(z)\, \nabla_\phi q_\phi(z) - \nabla_\phi q_\phi(z) \log q_\phi(z)\Big)\, dz$$

$$= \int \Big(\log h(z)\, \nabla_\phi q_\phi(z) - q_\phi(z) \nabla_\phi \log q_\phi(z) - \log q_\phi(z)\, \nabla_\phi q_\phi(z)\Big)\, dz.$$

$$= \int \log h(z)\, q_\phi(z) \nabla_\phi \log q_\phi(z) - q_\phi(z) \nabla_\phi \log q_\phi(z) - \left(\log q_\phi(z)\right) q_\phi(z) \nabla_\phi \log q_\phi(z)\, dz$$

$$= \int q_\phi(z) \nabla_\phi \log q_\phi(z) \Big(\log h(z) - 1 - \log q_\phi(z)\Big)\, dz$$

$$= \int q_\phi(z) \Big(\nabla_\phi \log q_\phi(z)\Big) \log\left(\frac{h(z)}{q_\phi(z)}\right) dz - \int q_\phi(z) \nabla_\phi \log q_\phi(z)\, dz$$

$$= \mathbb{E}_{z \sim q_\phi(z)}\left[\Big(\nabla_\phi \log q_\phi(z)\Big) \log\left(\frac{h(z)}{q_\phi(z)}\right)\right] - \int \nabla_\phi q_\phi(z)\, dz$$

$$= \mathbb{E}_{z \sim q_\phi(z)}\left[\Big(\nabla_\phi \log q_\phi(z)\Big) \log\left(\frac{h(z)}{q_\phi(z)}\right)\right]$$

(* Since $\int \nabla_\phi q_\phi(z)\, dz = \nabla_\phi \int q_\phi(z)\, dz = \nabla_\phi(1) = 0$)

#2 Project gradient method.

$$\begin{cases} \underset{x \in \mathbb{R}^n}{\text{maximize}}\ f(x) \\ \text{subject to } x \in C \quad (C \subset \mathbb{R}^n) \end{cases}$$

projected gradient : $x^{k+1} = \Pi_C(x^k - \alpha \nabla f(x^k))$ $\left(\Pi_C : \text{projection onto } C \quad \Pi_C(y) = \underset{x \in C}{\text{argmin}} \|x-y\|^2\right)$

Consider $C = \{ x \in \mathbb{R}^2 \mid x_1 = a,\ 0 \le x_2 \le 1 \}$ $\quad x = (x_1, x_2)$

Show $\Pi_C(y) = \begin{bmatrix} a \\ \min\{\max\{y_2, 0\}, 1\} \end{bmatrix}$ $\quad y = (y_1, y_2)$

pf) Since C only consists of $x \in \mathbb{R}^2$ s.t $x_1 = a$, $(\Pi_C(y))_1$ should be a.

$$\|x-y\|^2 = \|x_1 - y_1\|^2 + \|x_2 - y_2\|^2 = \underbrace{\|a - y_1\|^2}_{\text{just fixed value}} + \|x_2 - y_2\|^2.$$ So $\underset{x \in C}{\text{argmin}} \|x-y\|^2 = \left(a,\ \underset{x_2 \in [0,1]}{\text{argmin}} \|x_2 - y_2\|^2\right)$

$$\underset{x_2 \in [0,1]}{\text{argmin}} \|x_2 - y_2\|^2 = \begin{cases} 1 & \text{if } y_2 \ge 1 \\ y_2 & \text{if } 0 < y_2 < 1 \\ 0 & \text{if } 0 \ge y_2 \end{cases} = \begin{cases} 1 & \text{if } y_2 \ge 1 \\ \max(y_2, 0) & \text{if } y_2 < 1 \end{cases} = \min\{\max\{y_2, 0\}, 1\}$$

P3. 몇줄만 추가해주면 된다.

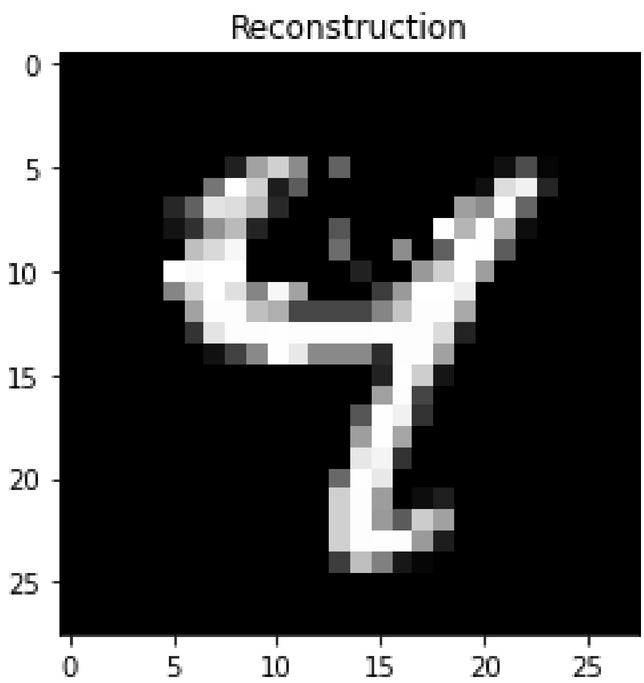
flow_inpainting.py

```python
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

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

##############################################
# STEP 1: Define dataset and preprocessing #
##############################################


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)

####################################
# STEP 3: Implement Coupling Layer #
####################################

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), nn.ReLU())
                                                            for _ in range(hidden - 1)])
        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_)
```

```python
        shift = self.out_block(off_)

        if reverse:
            on = on - shift
        else:
            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

##########################
# STEP 4: Implement NICE #
##########################

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
```

```python
    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)


# Load pre-trained NICE model onto CPU
model = NICE(in_out_dim=784, mid_dim=1000, hidden=5).to(device)
model.load_state_dict(torch.load('nice.pt',map_location=torch.device('cpu')))

# Since we do not update model, set requires_grad = False
model.requires_grad_(False)

# Get an MNIST image
testset = torchvision.datasets.MNIST(root='./', train=False, download=True, transform=torchvision.transforms.ToTensor())
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')


lr = 1e-3
recon = image.clone().requires_grad_(True)

for i in range(300):
    loss = -model(recon.view(1,28*28))
    loss.backward()
    recon.data = mask*recon.data + torch.clamp(recon.data - lr * recon.grad,0,1) *(~mask)
```

```python
# Plot reconstruction
plt.figure(figsize = (4,4))
plt.title('Reconstruction')
plt.imshow(make_grid(recon.squeeze().detach()).permute(1,2,0))
# plt.show()
plt.savefig('plt3.png')
```



#4

$A = PL(U + \mathrm{diag}(s)) \in \mathbb{R}^{c\times c}$

(a) $f_1(x) = Ax$.

$\left(\frac{\partial f_1}{\partial x}\right) = A$, $\quad \left|\frac{\partial f_1}{\partial x}\right| = |\det A| = |\det P||\det L||\det U + \mathrm{diag}(s)| = 1 \times 1 \times \left|\prod_{i=1}^{c} s_i\right| \Rightarrow \log\left|\frac{\partial f_1}{\partial x}\right| = \sum_{i=1}^{c} \log|s_i|$.

(We showed $\det P = 1$ in previous homework, and we know determinant of upper/lower triangular matrix is just product of the diagonals)

(b) $h: \mathbb{R}^{a\times b\times c} \to \mathbb{R}^{a\times b\times c}$

서로 다른 reshape operation, reshape1 과 reshape2 가 주어졌다고 하자. 그러면 $Y.\mathrm{reshape1}(abc) = P \times Y.\mathrm{reshape2}(abc)$ 인 $(Y \in \mathbb{R}^{a\times b\times c})$ permutation matrix $P \in \mathbb{R}^{abc\times abc}$ 가 존재한다.

$\therefore \left|\frac{\partial(h(X).\mathrm{reshape1}(abc))}{\partial(X.\mathrm{reshape1}(abc))}\right| = \left|\frac{\partial(P\,h(X).\mathrm{reshape2}(abc))}{\partial(X.\mathrm{reshape1}(abc))}\right| = \left|\frac{\partial(h(X).\mathrm{reshape2}(abc))}{\partial(X.\mathrm{reshape1}(abc))}\right|$ Since changing rows only change the "sign" of determinant.

$= \left|\frac{\partial(h(X).\mathrm{reshape2}(abc))}{\partial(P\,X.\mathrm{reshape2}(abc))}\right| = \left|\frac{\partial(h(X).\mathrm{reshape2}(abc))}{\partial(X.\mathrm{reshape2}(abc))}\right|$ Since changing columns only change the sign of determinant!

$\therefore \left|\frac{\partial h(X)}{\partial X}\right|$ does not depend on how we reshape!

(c) $f_2(X|P,L,U,s) : \mathbb{R}^{c\times m\times n} \to \mathbb{R}^{c\times m\times n}$, $\quad w \in \mathbb{R}^{c\times c\times 1\times 1}$, $\quad w_{i,j,1,1} = A_{ij}$

$\left|\frac{\partial f_2(X|P,L,U,s)}{\partial X}\right| = \left|\frac{\partial(f_2(X).\mathrm{reshape}(cmn))}{\partial(X.\mathrm{reshape}(cmn))}\right|$

$$\left(\frac{\partial(f_2(X).\mathrm{reshape}(cmn))}{\partial(X.\mathrm{reshape}(cmn))}\right) = \begin{pmatrix} \boxed{A} & & & & 0 \\ & \boxed{A} & & & \\ & & \boxed{A} & & \\ 0 & & & \ddots & \\ & & & & \boxed{A} \end{pmatrix} \in \mathbb{R}^{cmn\times cmn}$$

1X1 convolution 이므로
적절한 reshaping 을 생각하면 이런 식으로 Jacobian이 구성됨을 알 수 있다.

$\therefore \log\left|\frac{\partial f_2(X)}{\partial X}\right| = \log\left(|\det A|^{mn}\right) = mn\log|\det A| = mn\sum_{i=1}^{c}\log|s_i|$ //.

(d) $\begin{cases} Z_{1:c,:,:} = X_{1:c,:,:} \\ Z_{c+1:2c,:,:} = f_2(X_{c+1:2c,:,:} \mid P, L(X_{1:c,:,:}), U(X_{1:c,:,:}), s(X_{1:c,:,:})) \end{cases}$, $\quad X, Z \in \mathbb{R}^{2c\times m\times n}$

$\left|\frac{\partial Z}{\partial X}\right| = \left|\frac{\partial(Z.\mathrm{reshape}(2cmn))}{\partial(X.\mathrm{reshape}(2cmn))}\right|$

$$\frac{\partial(Z.\mathrm{reshape}(2cmn))}{\partial(X.\mathrm{reshape}(2cmn))} = \left(\begin{array}{c|c} \begin{matrix} 1 & & 0 \\ & 1 & \\ 0 & & \ddots \;1 \end{matrix} & 0 \\ \hline * & \underbrace{\begin{matrix} A & & & \\ & A & & \\ & & \boxed{A} & \\ & & & \ddots\; A \end{matrix}}_{cmn} \end{array}\right) \in \mathbb{R}^{2cmn\times 2cmn}$$

(*: 알 수 없는 복잡한 놈)

$\Rightarrow \left|\frac{\partial(Z.\mathrm{reshape}(2cmn))}{\partial(X.\mathrm{reshape}(2cmn))}\right| = |\det I_{cmn}||\det A|^{mn} = |\det A|^{mn}$

$\therefore \log\left|\frac{\partial Z}{\partial X}\right| = mn\log|\det A| = mn\sum_{i=1}^{c}\log|s_i|$

P5. 잘 나온다.

Gambler's_ruin.py* ×

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Nov 14 21:00:29 2022

@author: sylee
"""

import numpy as np

p = 18/37
q = 0.55

def phi(X):
    curr = 100
    for i in range(len(X)):
        if X[i]==1:
            curr += 1
        else:
            curr -= 1
        if curr==200:
            return 1
    return 0

def sample(prob, K):
    X=[]
    for i in range(K):
        n = np.random.rand()
        if n<prob:
            X.append(1)
        else:
            X.append(0)
    return X

def f(X, prob):
    answer =1
    for i in range(len(X)):
        if X[i]==1:
            answer *= prob
        else:
            answer *= (1-prob)
    return answer

def estimate(N, K, sampling_prob, real_prob):
    summation = 0
    for _ in range(N):
        X = sample(sampling_prob, K)
        summation += phi(X) * f(X, real_prob)/f(X, sampling_prob)
    return summation/N


print(estimate(3000, 600, q, p))
```

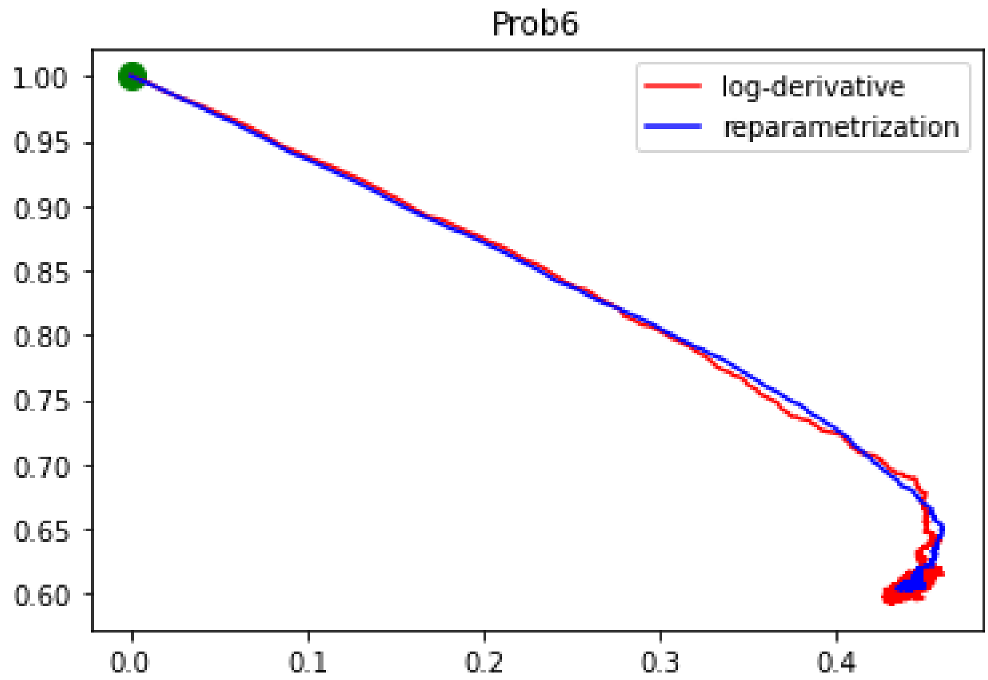
```
In [12]: runfile('C:/Users/sylee/OneDrive/
Users/sylee/OneDrive/바탕 화면/실수기/HW10
2.01061199926214e-06
```

P6. 잘 나온다.

SGD.py* ×

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 15 00:41:40 2022

@author: sylee
"""
import torch
import numpy as np
import matplotlib.pyplot as plt

lr = 1e-2
B = 300
iterations = 10000

#log derivative
theta = torch.tensor([0., 0.])
history1 = torch.zeros((iterations+1, 2))
history1[0][1] = 1

for itr in range(iterations):
    mu,tau  = theta[0], theta[1]
    sigma = tau.exp()

    X = torch.normal(mu,sigma,size=(B,1))
    g = torch.tensor([mu-1, sigma -1])
    g1 = torch.mean((X * X.sin()) * (X-mu)/sigma**2)
    g2 = torch.mean((X * X.sin()) * (-1+(X-mu)**2/sigma**2))
    g = g+torch.tensor([g1,g2])
    theta -= lr*g

    history1[itr+1] = theta
    history1[itr+1][1] = history1[itr+1][1].exp()
print(mu,sigma)

#reparametrization trick
theta = torch.tensor([0., 0.])
history2 = torch.zeros((iterations+1, 2))
history2[0][1] = 1

for itr in range(iterations):
    mu,tau = theta[0], theta[1]
    sigma = tau.exp()

    Y = torch.normal(0,1,size=(B,1))
    X = sigma * Y + mu

    g = torch.tensor([mu-1, sigma -1])
    g1 = torch.mean(X.sin() + X * X.cos())
    g2 = torch.mean( (X.sin() + X * X.cos())*Y*sigma )
    g = g+torch.tensor([g1,g2])

    theta -= lr*g

    history2[itr+1] = theta
    history2[itr+1][1] = history2[itr+1][1].exp()

print(mu, sigma)
```

```python
x1 = np.array(history1[:, 0])
y1 = np.array(history1[:, 1])
x2 = np.array(history2[:, 0])
y2 = np.array(history2[:, 1])
plt.scatter(0,1, s=100, c='green')
plt.plot(x1, y1, linestyle='solid',color='red', label = 'log-derivative')
plt.plot(x2, y2, linestyle='solid',color='blue', label = 'reparametrization')
plt.title('Prob6')
plt.legend()
plt.show()
```




```
In [25]: runfile('C:/Users/sylee
OneDrive/바탕 화면/실수기/HW10')
tensor(0.4430) tensor(0.6080)
tensor(0.4435) tensor(0.6063)
```