MathDNN Homework 12

Department of Computer Science and Engineering
2021-16988 Jaewan Park

Problem 2

Let $\mathcal{L}(\theta, \phi)$ the objective of the minimax problem, then

$$\begin{aligned}\mathcal{L}(\theta, \phi) :&= \mathbb{E}_{X \sim p_{\text{true}}}[\log D_\phi(X)] + \lambda \mathbb{E}_{\tilde{X} \sim p_\theta}\left[\log\left(1 - D_\phi\left(\tilde{X}\right)\right)\right] \\ &= \int \Big(p_{\text{true}}(x) \log D_\phi(x) + \lambda p_\theta(x) \log\left(1 - D_\phi(x)\right)\Big) dx.\end{aligned}$$

Since

$$\frac{d}{dy}(a \log y + b \log(1-y)) = \frac{a}{y} - \frac{b}{1-y} = 0 \Leftrightarrow y = \frac{a}{a+b}$$

we can say $\mathcal{L}(\theta, \phi)$ is maximized by

$$D_\phi(x) = \frac{p_{\text{true}}(x)}{p_{\text{true}}(x) + \lambda p_\theta(x)}.$$

and let the corresponding ϕ as ϕ^*. Then we obtain

$$\begin{aligned}\max_{\phi \in \mathbb{R}^p} \mathcal{L}(\theta, \phi) &= \mathcal{L}(\theta, \phi^*) \\ &= \int \left(p_{\text{true}}(x) \log\left(\frac{p_{\text{true}}(x)}{p_{\text{true}}(x) + \lambda p_\theta(x)}\right) + \lambda p_\theta(x) \log\left(\frac{\lambda p_\theta(x)}{p_{\text{true}}(x) + \lambda p_\theta(x)}\right)\right) dx \\ &= D_f(p_{\text{true}} \,||\, p_\theta) - \Big((1+\lambda)\log(1+\lambda) - \lambda \log \lambda\Big).\end{aligned}$$

Therefore we can say that the following problems are equivalent.

$$\begin{aligned}\underset{\theta \in \mathbb{R}^p}{\text{minimize}}\ \underset{\phi \in \mathbb{R}^p}{\text{maximize}}\ \mathcal{L}(\theta, \phi) &\Leftrightarrow \underset{\theta \in \mathbb{R}^p}{\text{minimize}} \Big(D_f(p_{\text{true}} \,||\, p_\theta) - \Big((1+\lambda)\log(1+\lambda) - \lambda \log \lambda\Big)\Big) \\ &\Leftrightarrow \underset{\theta \in \mathbb{R}^p}{\text{minimize}}\ D_f(p_{\text{true}} \,||\, p_\theta)\end{aligned}$$

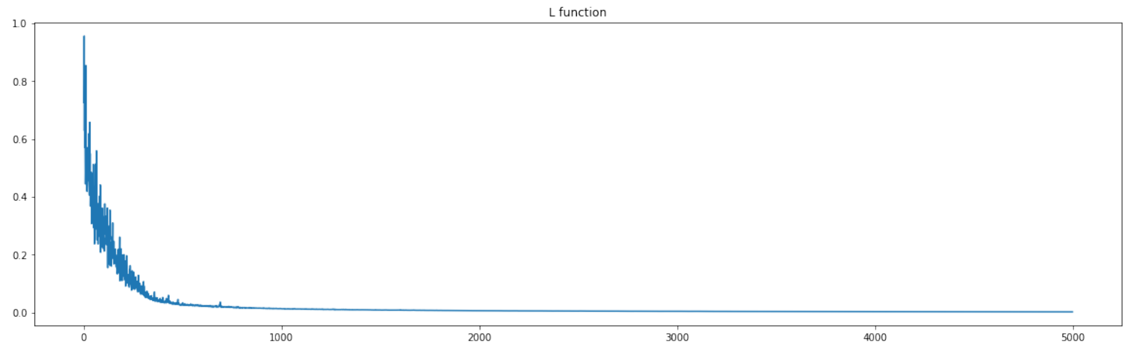
Problem 1

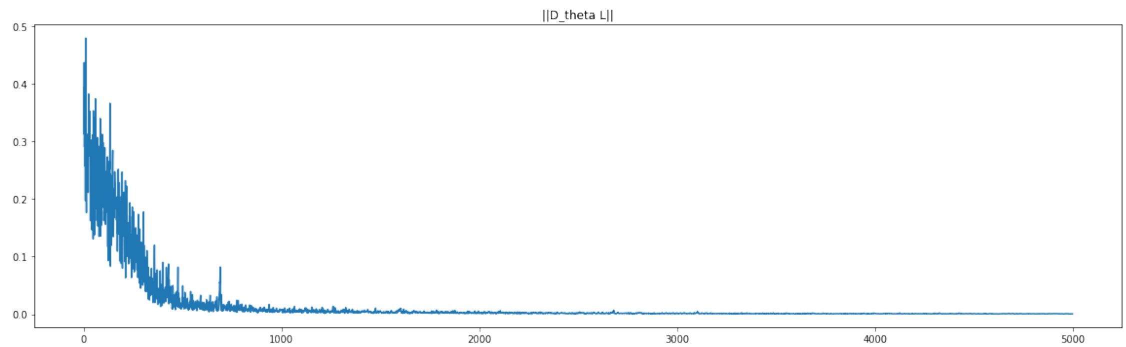
[1]:
```
import numpy as np
from matplotlib import pyplot as plt

N, p = 30, 20
np.random.seed(0)
X = np.random.randn(N,p)
Y = 2*np.random.randint(2, size = N) - 1
lamda = 0.001
```

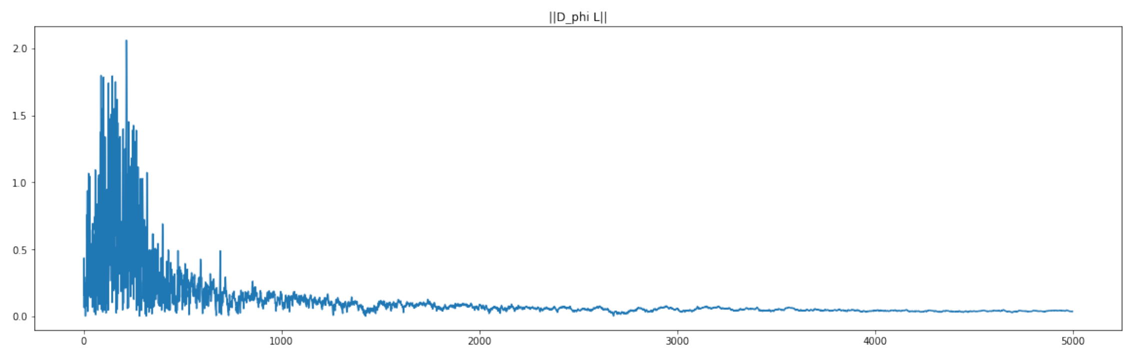
[2]:
```
theta = 0.1 * np.random.randn(p)
phi = 0.1 * np.random.randn(p)
alpha = 3e-1
beta = 1e-4

epoch = 5000
L_val = []
d_phi_val = []
d_theta_val = []

for _ in range(epoch):
    for __ in range(N):
        ind = np.random.randint(N)
        phi += beta * Y[ind] * theta * np.exp(-Y[ind] * np.inner((X[ind, :] - phi),␣
↪theta)) / (1 + np.exp(-Y[ind] * np.inner((X[ind, :] - phi), theta))) - lamda * phi
        theta -= alpha * (-Y[ind] * (X[ind, :] - phi)) * np.exp(-Y[ind] * np.
↪inner((X[ind, :] - phi), theta)) / (1 + np.exp(-Y[ind] * np.inner((X[ind, :] - phi),␣
↪theta)))

    L_i = np.average(np.log(1 + np.exp(-Y * ((X - phi.reshape(1,-1)) @ theta)))) -␣
↪lamda/2 * np.linalg.norm(phi, axis=0, ord=2) **2
    d_phi = np.average(Y / (1 + np.exp(Y * ((X-phi.reshape(1,-1)) @ theta)))) * theta -␣
↪lamda * phi
    d_theta = np.average(( -Y / (1 + np.exp(Y * ((X-phi.reshape(1,-1)) @ theta))) ).
↪reshape(-1,1)*(X-phi.reshape(1,-1)), axis=0)

    L_val.append(L_i)
    d_phi_val.append(d_phi)
    d_theta_val.append(d_theta)
```

[3]:
```
fig, ax = plt.subplots(figsize=(20, 20))
plt.subplots_adjust(left=0.125,
                    bottom=0.1,
                    right=0.9,
                    top=0.9,
                    wspace=0.2,
                    hspace=0.35)
plt.subplot(3, 1, 1)
plt.title("L function")
plt.plot(L_val)
plt.subplot(3, 1, 2)
plt.title("||D_theta L||")
```

```
plt.plot(np.linalg.norm(d_theta_val, axis=1, ord=2))
plt.subplot(3, 1, 3)
plt.title("||D_phi L||")
plt.plot(np.linalg.norm(d_phi_val, axis=1, ord=2))
plt.show()
```







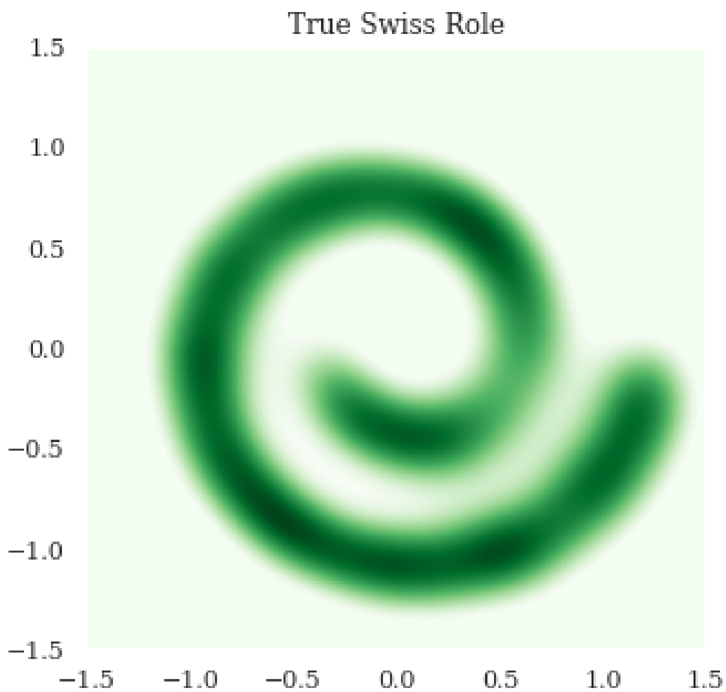
Swiss Roll

[4]:
```
import torch
import torch.nn as nn
from torch.utils.data import Dataset, TensorDataset, DataLoader
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

[5]:
```
"""
Step 0 : Define training configurations
"""

batch_size = 64
learning_rate = 5e-4
num_epochs = 2000
reg_coeff = 75
device = "cuda:0" if torch.cuda.is_available() else "cpu"
```

[6]:
```
"""
Step 1 : Define custom dataset
"""

def make_swiss_roll(n_samples=2000, noise = 1.0, dimension = 2, a = 20, b = 5):
    """
    Generate 2D swiss roll dataset
    """
    t = 2 * np.pi * np.sqrt(np.random.uniform(0.25,4,n_samples))

    X = 0.1 * t * np.cos(t)
    Y = 0.1 * t * np.sin(t)

    errors = 0.025 * np.random.multivariate_normal(np.zeros(2), np.eye(dimension), size␣
↪= n_samples)
    X += errors[:, 0]
    Y += errors[:, 1]
    return np.stack((X, Y)).T

def show_data(data, title):
    """
    Plot the data distribution
    """
    sns.set(rc={'axes.facecolor': 'honeydew', 'figure.figsize': (5.0, 5.0)})
    plt.figure(figsize = (5, 5))
    plt.rc('text', usetex = False)
    plt.rc('font', family = 'serif')
    plt.rc('font', size = 10)

    g = sns.kdeplot(x=data[:, 0], y=data[:, 1], fill=True, thresh=0.1, levels=1000,␣
↪cmap="Greens")

    g.grid(False)
    plt.margins(0, 0)
```

```
    plt.xlim(-1.5,1.5)
    plt.ylim(-1.5,1.5)
    plt.title(title)
    plt.show()

data = make_swiss_roll()
show_data(data, 'True Swiss Role')
```



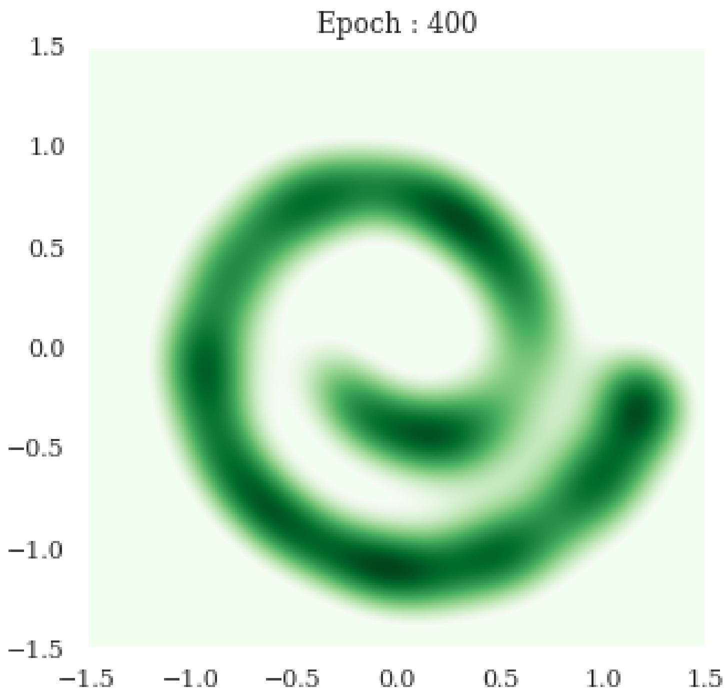
[7]:
```
"""
Step 2 : Define custom dataset and dataloader.
"""

class SwissRollDataset(Dataset) :
    def __init__(self, data) :
        super().__init__()
        self.data = torch.from_numpy(data)

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

    def __getitem__(self, idx) :
        return self.data[idx]

dataset = SwissRollDataset(data)
loader = DataLoader(dataset, batch_size = batch_size, shuffle = True)
```

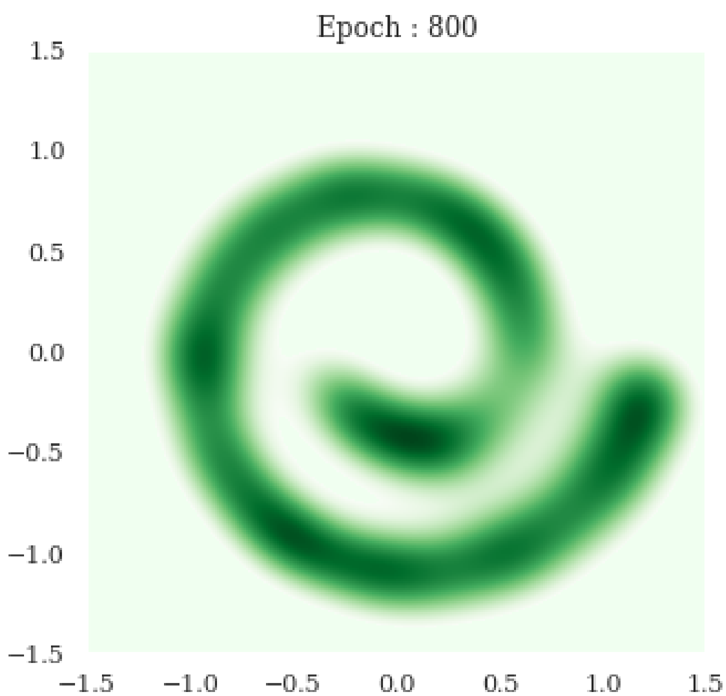
Problem 3 : Using VAE

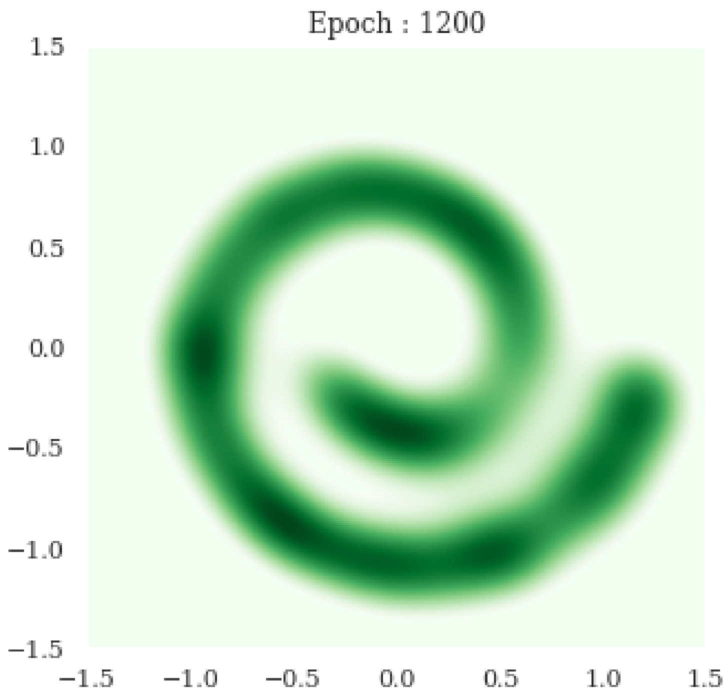
[8]:
```
"""
Step 3 : Implement models
"""

latent_dim = 16

class Encoder(nn.Module):
    def __init__(self):
        super(Encoder,self).__init__()
        self.fc = nn.Sequential(nn.Linear(2, 128),
                                nn.LeakyReLU(0.2),
                                nn.Linear(128, 128),
                                nn.Tanh(),
                                nn.Linear(128, 2*latent_dim))

    def forward(self,x):
        z = self.fc(x)
        mu = z[:, :latent_dim]
        log_std = z[:, latent_dim:]
        return mu, log_std

class Decoder(nn.Module):
    def __init__(self):
        super(Decoder,self).__init__()
        self.fc = nn.Sequential(nn.Linear(latent_dim, 64),
                                nn.LeakyReLU(0.2),
                                nn.Linear(64, 64),
                                nn.Tanh(),
                                nn.Linear(64, 2))

    def forward(self, z):
        return self.fc(z)

# Instantiating Encoder and Decoder
Encoder = Encoder().to(device)
Decoder = Decoder().to(device)

# optimizer
optimizer = torch.optim.Adam(list(Encoder.parameters()) + list(Decoder.parameters()),␣
↪lr=learning_rate)

# log_p_theta_(z|x) (reconstruction loss)
def log_p(x_hat, x):
    mse = nn.MSELoss().to(device)
    return -mse(x_hat, x)

# reparametrization trick
def reparametrization_trick(mu, log_std):
    z = torch.randn_like(mu)*log_std.exp()+mu
    return z
```

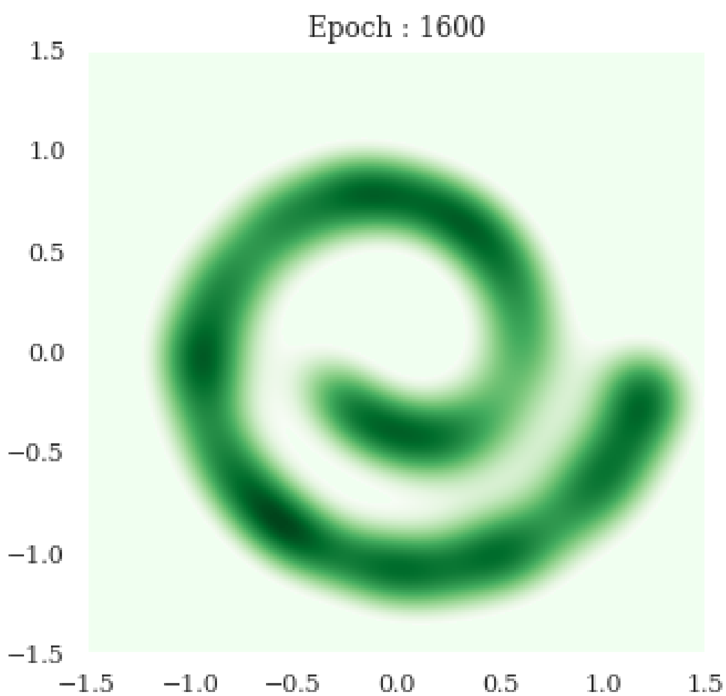
```python
def kl_div(mu, log_std):
    kl = -log_std - 0.5 + (torch.exp(2 * log_std) + mu ** 2) * 0.5
    kl = kl.sum(1).mean()
    return kl
```

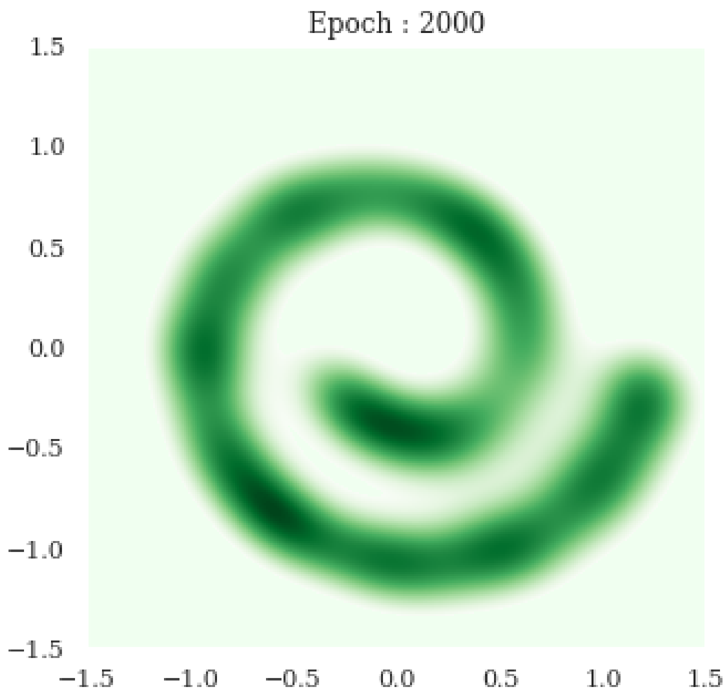
[9]:
```python
"""
Step 4 : Train models
"""
for epoch in range(1, num_epochs + 1) :
    for batch_idx, x in enumerate(loader) :
        x = x.float().detach().to(device)
        mu, log_std = Encoder(x)
        z = reparametrization_trick(mu, log_std)
        x_hat = Decoder(z)

        recon_loss = -log_p(x_hat, x)
        regularizer = kl_div(mu, log_std)

        loss = recon_loss * reg_coeff + regularizer

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    # Visualize the intermediate result
    if epoch % (num_epochs // 5) == 0:
        data = torch.Tensor(0, 2)
        for batch_idx, x in enumerate(loader) :
            x = x.float().detach().to(device)
            mu, log_std = Encoder(x)
            x_hat = Decoder(mu)
            data = torch.cat((data, x_hat.data), 0)

        show_data(data, f"Epoch : {epoch}")
```


Epoch : 400
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Epoch : 800
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Epoch : 1200
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Epoch : 1600
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

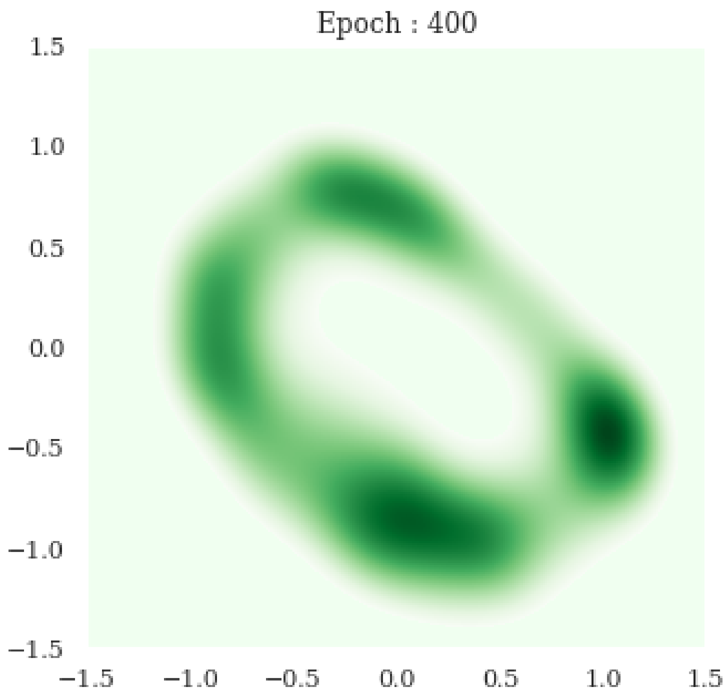


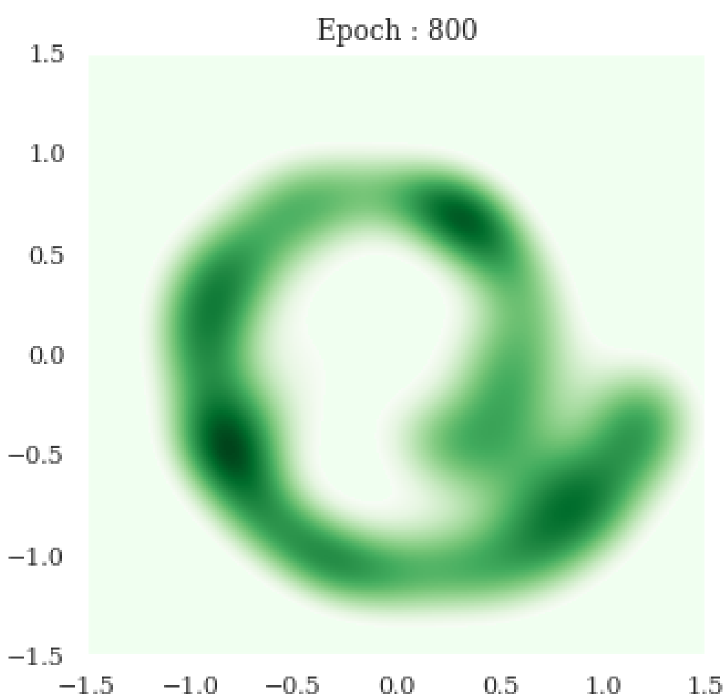
Problem 4 : Using GAN

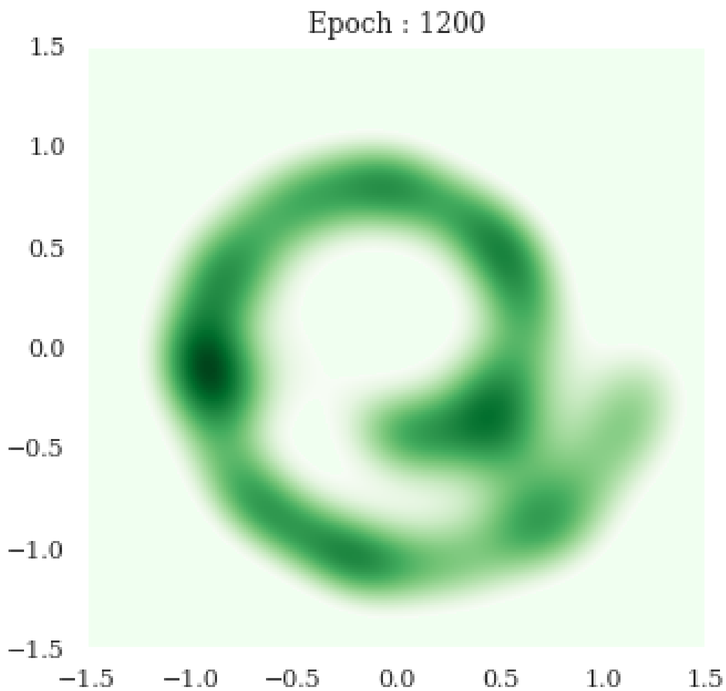
[16]:
```
"""
Step 3 : Implement models
"""

class Generator(nn.Module):
    def __init__(self, g_input_dim, g_output_dim):
        super(Generator, self).__init__()
        self.fc = nn.Sequential(nn.Linear(g_input_dim, 32),
                                nn.Tanh(),
                                nn.Linear(32, g_output_dim))

    def forward(self, x):
        return self.fc(x)

class Discriminator(nn.Module):
    def __init__(self, d_input_dim):
        super(Discriminator, self).__init__()
        self.fc = nn.Sequential(nn.Linear(d_input_dim, 128),
                                nn.Tanh(),
                                nn.Linear(128, 128),
                                nn.Tanh(),
                                nn.Linear(128, 1),
                                nn.Sigmoid())

    def forward(self, x):
```

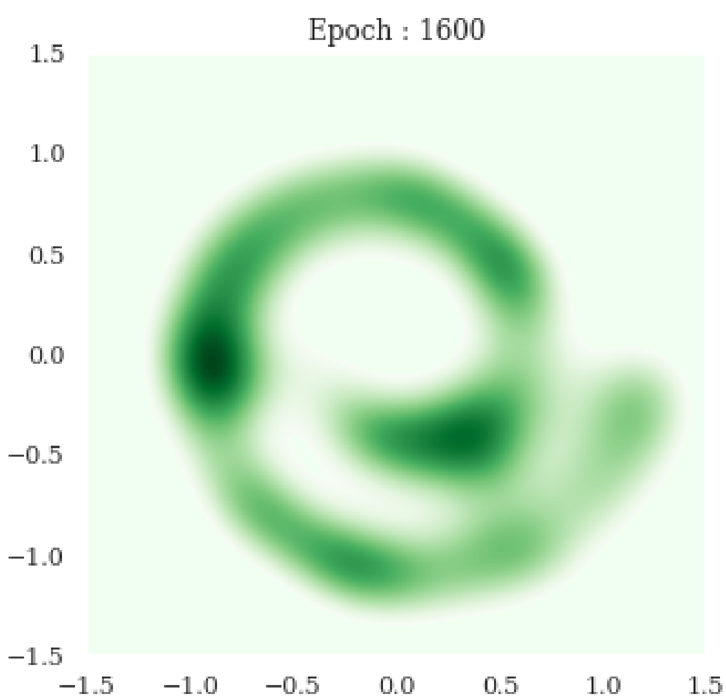
```
        return self.fc(x)

z_dim = 100

G = Generator(z_dim, 2).to(device)
D = Discriminator(2).to(device)

G_optimizer = torch.optim.Adam(G.parameters(), lr = learning_rate)
D_optimizer = torch.optim.Adam(D.parameters(), lr = learning_rate)
```

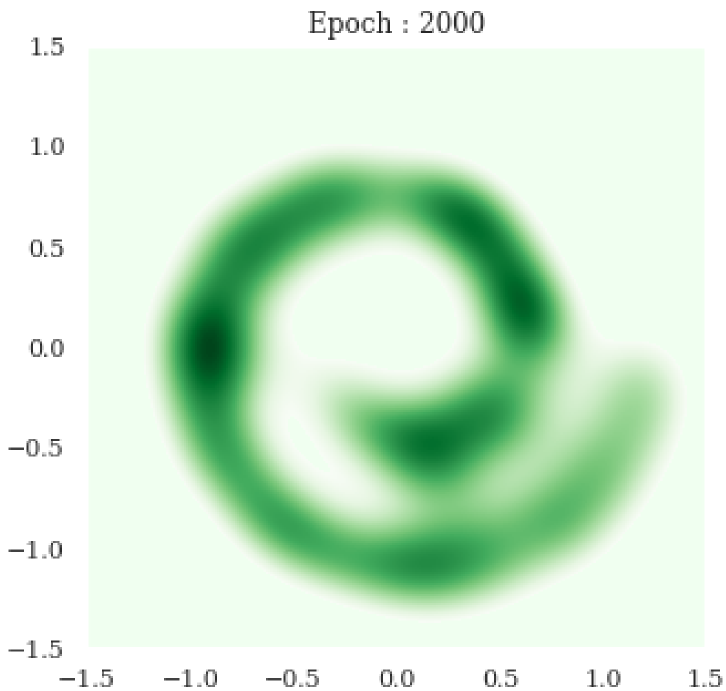
[17]:
```
"""
Step 4 : Train models
"""

for epoch in range(1, num_epochs + 1) :
    for batch_idx, x in enumerate(loader) :
        D.zero_grad()
        x = x.float().to(device)
        D_real_loss = torch.mean(torch.log(D(x)))
        z = torch.randn(batch_size, z_dim).to(device)
        D_fake_loss = torch.mean(torch.log(1-D(G(z))))
        D_loss = -(D_real_loss + 500 * D_fake_loss)
        D_loss.backward()
        D_optimizer.step()

        G.zero_grad()
        z = torch.randn(batch_size, z_dim).to(device)
        G_loss = -torch.mean(torch.log(D(G(z))))
        G_loss.backward()
        G_optimizer.step()

    # Visualize the intermediate result
    if epoch % (num_epochs // 5) == 0:
        with torch.no_grad():
            data = torch.Tensor(0, 2)
            for batch_idx, x in enumerate(loader) :
                generated = G(torch.randn(batch_size, z_dim).to(device))
                data = torch.cat((data, generated.data), 0)

            show_data(data, f"Epoch : {epoch}")
```


Epoch : 400
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Epoch : 800
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Epoch : 1200
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Epoch : 1600
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Epoch : 2000
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5