MathDNN Homework 9

Department of Computer Science and Engineering
2021-16988 Jaewan Park

Problem 3

Consider Ω and $\Omega^\complement$ as ordered and sorted sets. Now define f and g as $f(x) = i|_{x \text{ is the } i\text{th element of } \Omega}$ and $g(y) = j|_{y \text{ is the } j\text{th element of } \Omega^\complement}$ for $x \in \Omega$ and $y \in \Omega^\complement$ each. We can calculate the Jacobian matrix between the layers in the form of

$$\frac{\partial z}{\partial x} = \left\{ \frac{\partial z_i}{\partial x_j} \right\}_{i,j}, \quad \frac{\partial z_i}{\partial x_j} = \begin{cases} 1 & \left(i \in \Omega,\ i = j\right) \\ \frac{\partial [s_\theta(x_\Omega)]_{g(i)}}{\partial x_j} e^{[s_\theta(x_\Omega)]_{g(i)}} x_i + \frac{\partial [t_\theta(x_\Omega)]_{g(i)}}{\partial x_j} & \left(i \in \Omega^\complement,\ j \in \Omega\right) \\ e^{[s_\theta(x_\Omega)]_{g(i)}} \left(= e^{[s_\theta(x_\Omega)]_{g(j)}}\right) & \left(i \in \Omega^\complement,\ j \in \Omega^\complement,\ i = j\right) \\ 0 & \left(\text{otherwise}\right) \end{cases}.$$

Selecting σ such that $\sigma^{-1}(i) = \begin{cases} f^{-1}(i) & (i \leq |\Omega|) \\ g^{-1}(i - |\Omega|) & (i > |\Omega|) \end{cases}$ gives

$$\begin{aligned} P_\sigma \frac{\partial z}{\partial x} P_{\sigma^{-1}} &= \begin{bmatrix} \partial z_{\sigma^{-1}(1)} / \partial x_{\sigma^{-1}(1)} & \partial z_{\sigma^{-1}(1)} / \partial x_{\sigma^{-1}(2)} & \cdots & \partial z_{\sigma^{-1}(1)} / \partial x_{\sigma^{-1}(n)} \\ \partial z_{\sigma^{-1}(2)} / \partial x_{\sigma^{-1}(1)} & \partial z_{\sigma^{-1}(2)} / \partial x_{\sigma^{-1}(2)} & \cdots & \partial z_{\sigma^{-1}(2)} / \partial x_{\sigma^{-1}(n)} \\ \vdots & \vdots & \ddots & \vdots \\ \partial z_{\sigma^{-1}(n)} / \partial x_{\sigma^{-1}(1)} & \partial z_{\sigma^{-1}(n)} / \partial x_{\sigma^{-1}(2)} & \cdots & \partial z_{\sigma^{-1}(n)} / \partial x_{\sigma^{-1}(n)} \end{bmatrix} \\ &= \begin{bmatrix} I & 0 \\ * & \operatorname{diag}\left(e^{s_\theta(x_\Omega)}\right) \end{bmatrix}. \end{aligned}$$

Therefore $\dfrac{\partial z}{\partial x}$ can be decomposed in the form of

$$\frac{\partial z}{\partial x} = P_{\sigma^{-1}} \begin{bmatrix} I & 0 \\ * & \operatorname{diag}\left(e^{s_\theta(x_\Omega)}\right) \end{bmatrix} P_\sigma,$$

and we can calculate the determinant as

$$\begin{aligned} \log \left| \frac{\partial z}{\partial x} \right| &= \log \begin{vmatrix} I & 0 \\ * & \operatorname{diag}\left(e^{s_\theta(x_\Omega)}\right) \end{vmatrix} \\ &= \log \prod_{i \in \Omega^\complement} e^{[s_\theta(x_\Omega)]_{g(i)}} = \sum_{i \in \Omega^\complement} [s_\theta(x_\Omega)]_{g(i)} \\ &= \mathbf{1}_{n-|\Omega|}^\intercal s_\theta(x_\Omega). \end{aligned}$$

Problem 4

(a) Since $-\log$ is a convex function, we can apply Jensen's inequality to $-\log$, which gives

$$\begin{aligned} D_{\mathrm{KL}}(X||Y) &= \int_{\mathbb{R}^d} f(x)\log\left(\frac{f(x)}{g(x)}\right)dx = \mathbf{E}\left[\log\left(\frac{f(X)}{g(X)}\right)\right] = \mathbf{E}\left[-\log\left(\frac{g(X)}{f(X)}\right)\right] \\ &\geq -\log\left(\mathbf{E}\left[\frac{g(X)}{f(X)}\right]\right) = -\log\left(\int_{\mathbb{R}^d} f(x)\cdot\frac{g(x)}{f(x)}dx\right) = -\log 1 = 0. \end{aligned}$$

(b) Since $X_1, \cdots, X_d$ and $Y_1, \cdots, Y_d$ are each independent, when $f_1, \cdots, f_d$ and $g_1, \cdots, g_d$ are PDFs for $X_1, \cdots, X_d$ and $Y_1, \cdots, Y_d$ each, we can say

$$f(x) = f_1(x_1)\cdots f_d(x_d), \quad g(y) = g_1(y_1)\cdots g_d(y_d)$$

for any $x = (x_1, \cdots, x_d)$ and $y = (y_1, \cdots, y_d)$. Therefore

$$\begin{aligned} D_{\mathrm{KL}}(X||Y) &= \mathbf{E}\left[-\log\left(\frac{g(X)}{f(X)}\right)\right] = \mathbf{E}\left[-\log\left(\frac{g_1(X_1)}{f_1(X_1)}\right)\right] + \cdots + \mathbf{E}\left[-\log\left(\frac{g_d(X_d)}{f_d(X_d)}\right)\right] \\ &= D_{\mathrm{KL}}(X_1||Y_1) + \cdots + D_{\mathrm{KL}}(X_d||Y_d). \end{aligned}$$

Problem 5

The PDF of a multivariate Gaussian random variable $X \sim \mathcal{N}(\mu, \Sigma)$ with dimension d is given by

$$p_X(x) = \frac{1}{\sqrt{(2\pi)^d \det\Sigma}}\exp\left(-\frac{1}{2}(x-\mu)^\intercal\Sigma^{-1}(x-\mu)\right).$$

Let X_0, X_1 random variables that follow $\mathcal{N}(\mu_0, \Sigma_0)$, $\mathcal{N}(\mu_1, \Sigma_1)$ each. Also let their PDFs f_0, f_1. Then

$$\begin{aligned} &D_{\mathrm{KL}}(\mathcal{N}(\mu_0,\Sigma_0)\,||\mathcal{N}(\mu_1,\Sigma_1)) \\ &\qquad= \mathbf{E}\left[-\log\left(\frac{f_1(X_0)}{f_0(X_0)}\right)\right] = \mathbf{E}\Big[\log f_0(X_0) - \log f_1(X_0)\Big] \\ &\qquad= \mathbf{E}\left[\frac{1}{2}\log\frac{\det\Sigma_1}{\det\Sigma_0} - \frac{1}{2}(X_0-\mu_0)^\intercal\Sigma_0^{-1}(X_0-\mu_0) + \frac{1}{2}(X_0-\mu_1)^\intercal\Sigma_1^{-1}(X_0-\mu_1)\right] \\ &\qquad= \frac{1}{2}\log\frac{\det\Sigma_1}{\det\Sigma_0} - \frac{1}{2}\mathbf{E}\big[\mathrm{tr}\big((X_0-\mu_0)^\intercal\Sigma_0^{-1}(X_0-\mu_0)\big)\big] + \frac{1}{2}\mathbf{E}\big[(X_0-\mu_1)^\intercal\Sigma_1^{-1}(X_0-\mu_1)\big] \\ &\qquad= \frac{1}{2}\log\frac{\det\Sigma_1}{\det\Sigma_0} - \frac{1}{2}\mathbf{E}\big[\mathrm{tr}\big((X_0-\mu_0)(X_0-\mu_0)^\intercal\Sigma_0^{-1}\big)\big] + \frac{1}{2}\big((\mu_0-\mu_1)^\intercal\Sigma_1^{-1}(\mu_0-\mu_1) + \mathrm{tr}\big(\Sigma_1^{-1}\Sigma_0\big)\big) \\ &\qquad= \frac{1}{2}\log\frac{\det\Sigma_1}{\det\Sigma_0} - \frac{1}{2}\mathrm{tr}\big(\mathbf{E}[(X_0-\mu_0)(X_0-\mu_0)^\intercal]\Sigma_0^{-1}\big) + \frac{1}{2}\big((\mu_1-\mu_0)^\intercal\Sigma_1^{-1}(\mu_1-\mu_0) + \mathrm{tr}\big(\Sigma_1^{-1}\Sigma_0\big)\big) \\ &\qquad= \frac{1}{2}\log\frac{\det\Sigma_1}{\det\Sigma_0} - \frac{1}{2}\mathrm{tr}\big(\Sigma_0\Sigma_0^{-1}\big) + \frac{1}{2}\big((\mu_1-\mu_0)^\intercal\Sigma_1^{-1}(\mu_1-\mu_0) + \mathrm{tr}\big(\Sigma_1^{-1}\Sigma_0\big)\big) \\ &\qquad= \frac{1}{2}\left(\mathrm{tr}\big(\Sigma_1^{-1}\Sigma_0\big) + (\mu_1-\mu_0)^\intercal\Sigma_1^{-1}(\mu_1-\mu_0) - d + \log\left(\frac{\det\Sigma_1}{\det\Sigma_0}\right)\right). \end{aligned}$$

Problem 6

For each θ, let $\phi_\theta \in \Phi$ the value of ϕ that makes $h(\theta, \phi) = 0$. Then we obtain

$$\begin{aligned}
\sup_{\theta,\phi} g(\theta,\phi) &= \sup_\theta \left(\sup_\phi g(\theta,\phi) \right) \\
&= \sup_\theta \left(\sup_\phi \Big(f(\theta) - h(\theta,\phi) \Big) \right) = \sup_\theta \Big(f(\theta) - \inf_\phi h(\theta,\phi) \Big) \\
&= \sup_\theta f(\theta)
\end{aligned}$$

since $\inf_\phi h(\theta, \phi) = 0$, more precisely $\min_\phi h(\theta, \phi) = 0$ when $\phi = \phi_\theta$. Therefore we can conclude that

$$\operatorname{argmax} f = \{\theta \mid (\theta, \phi) \in \operatorname{argmax} g\}$$

and the two given optimization problems are equivalent.

Problem 1

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets
import torch.optim as optim
from torchvision.transforms import transforms
from torchvision.utils import save_image
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
lr = 0.001
batch_size = 100
epochs = 10
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

[3]:
```
'''
Step 1:
'''

# MNIST dataset
dataset = datasets.MNIST(root='./mnist_data/',
                         train=True,
                         transform=transforms.ToTensor(),
                         download=True)

train_dataset, validation_dataset = torch.utils.data.random_split(dataset, [50000,␣
↪10000])

test_dataset = datasets.MNIST(root='./mnist_data/',
                              train=False,
                              transform=transforms.ToTensor())

# KMNIST dataset, only need test dataset
anomaly_dataset = datasets.KMNIST(root='./kmnist_data/',
                                  train=False,
                                  transform=transforms.ToTensor(),
                                  download=True)

# print(len(train_dataset))  # 50000
# print(len(validation_dataset))  # 10000
# print(len(test_dataset))  # 10000
# print(len(anomaly_dataset))  # 10000
```

[4]:
```
'''
Step 2: AutoEncoder
'''
# Define Encoder
class Encoder(nn.Module):
    def __init__(self):
        super(Encoder, self).__init__()
```

```python
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 32)
    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        z = F.relu(self.fc3(x))
        return z

# Define Decoder
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self).__init__()
        self.fc1 = nn.Linear(32, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 784)
    def forward(self, z):
        z = F.relu(self.fc1(z))
        z = F.relu(self.fc2(z))
        x = torch.sigmoid(self.fc3(z))  # to make output's pixels are 0~1
        x = x.view(x.size(0), 1, 28, 28)
        return x
```

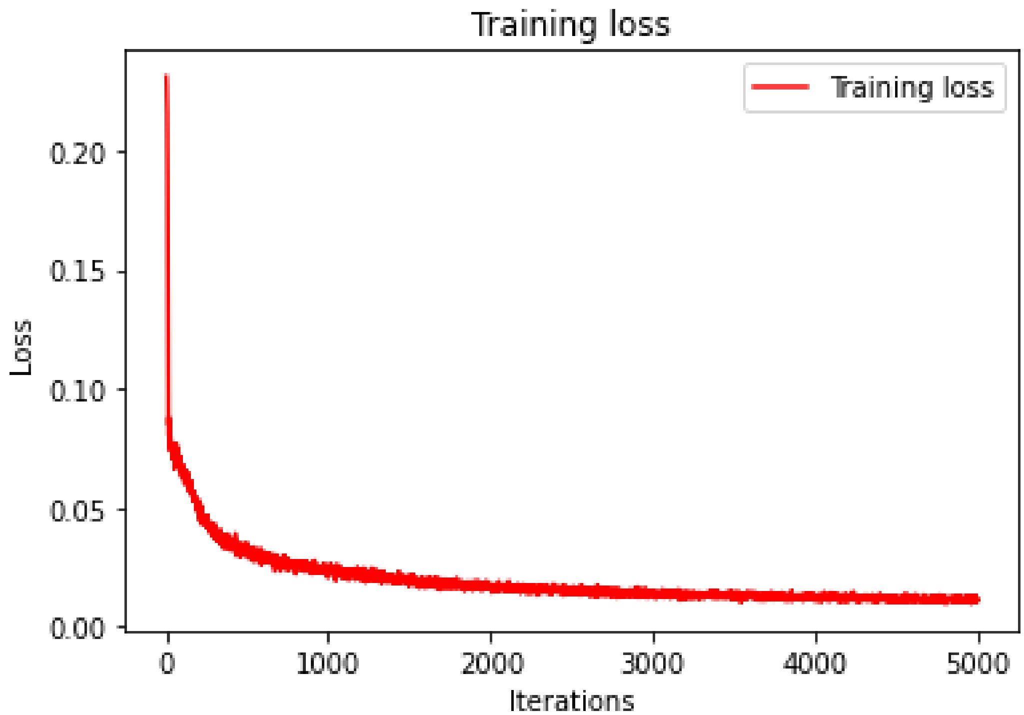
[5]:
```python
'''
Step 3: Instantiate model & define loss and optimizer
'''
enc = Encoder().to(device)
dec = Decoder().to(device)
loss_function = nn.MSELoss()
optimizer = optim.Adam(list(enc.parameters()) + list(dec.parameters()), lr=lr)
```

[6]:
```python
'''
Step 4: Training
'''
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)


train_loss_list = []

import time
start = time.time()
for epoch in range(epochs) :
    print("{}th epoch starting.".format(epoch))
    enc.train()
    dec.train()
    for batch, (images, _) in enumerate(train_loader) :
        images = images.to(device)
        z = enc(images)
        reconstructed_images = dec(z)

        optimizer.zero_grad()
```

```python
        train_loss = loss_function(images, reconstructed_images)
        train_loss.backward()
        train_loss_list.append(train_loss.item())

        optimizer.step()

        print(f"[Epoch {epoch:3d}] Processing batch #{batch:3d} reconstruction loss:␣
↪{train_loss.item():.6f}", end='\r')
end = time.time()
print("Time ellapsed in training is: {}".format(end - start))

# plotting train loss
plt.plot(range(1,len(train_loss_list)+1), train_loss_list, 'r', label='Training loss')
plt.title('Training loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.savefig('loss.png')

enc.eval()
dec.eval()
```

```
0th epoch starting.
1th epoch starting.ing batch #499 reconstruction loss: 0.030005
2th epoch starting.ing batch #499 reconstruction loss: 0.024107
3th epoch starting.ing batch #499 reconstruction loss: 0.021511
4th epoch starting.ing batch #499 reconstruction loss: 0.016795
5th epoch starting.ing batch #499 reconstruction loss: 0.014868
6th epoch starting.ing batch #499 reconstruction loss: 0.015011
7th epoch starting.ing batch #499 reconstruction loss: 0.011456
8th epoch starting.ing batch #499 reconstruction loss: 0.012015
9th epoch starting.ing batch #499 reconstruction loss: 0.013013
Time ellapsed in training is: 26.782079935073853 loss: 0.011306
```

[6]:
```
Decoder(
  (fc1): Linear(in_features=32, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=784, bias=True)
)
```



[7]:
```
'''
Step 5: Calculate standard deviation by using validation set
'''
validation_loader = torch.utils.data.DataLoader(dataset=validation_dataset,␣
 ↪batch_size=batch_size)

enc.eval()
dec.eval()

loss_list = []
for images, _ in validation_loader:
    images = images.to(device)
    z = enc(images)
    reconstructed_images = dec(z)
    errors = ((images - reconstructed_images)**2).view(batch_size, -1)
    for error in errors:
        loss_list.append(torch.mean(error))

mean = torch.mean(torch.tensor(loss_list))
std = torch.std(torch.tensor(loss_list))

threshold = mean + 3 * std
print("threshold: ", threshold)
```

```
threshold:  tensor(0.0329)
```

[8]:
```
'''
Step 6: Anomaly detection (mnist)
'''
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size)

type_I_error = 0
for images, _ in test_loader:
    images = images.to(device)
    z = enc(images)
    reconstructed_images = dec(z)
    errors = ((images - reconstructed_images)**2).view(batch_size, -1)
    for error in errors:
        if torch.mean(error) > threshold:
            type_I_error += 1

type_I_error_rate = type_I_error / (len(test_loader) * batch_size)
print("type 1 error rate : ", type_I_error_rate*100 , "%")
```

```
type 1 error rate :  1.04 %
```

[9]:
```
'''
Step 7: Anomaly detection (kmnist)
'''
anomaly_loader = torch.utils.data.DataLoader(dataset=anomaly_dataset,␣
↪batch_size=batch_size)

type_II_error = 0
for images, _ in anomaly_loader:
    images = images.to(device)
    z = enc(images)
    reconstructed_images = dec(z)
    errors = ((images - reconstructed_images)**2).view(batch_size, -1)
    for error in errors:
        if torch.mean(error) <= threshold:
            type_II_error += 1

type_II_error_rate = type_II_error / (len(anomaly_loader) * batch_size)
print("type 2 error rate : ", type_II_error_rate*100, "%")
```

```
type 2 error rate :  2.9499999999999997 %
```

Problem 2

[10]:
```
import torch
import torch.utils.data as data
import torch.nn as nn
from torch.distributions.normal import Normal
from torch.distributions.uniform import Uniform
import numpy as np
import matplotlib.pyplot as plt
```

[11]:
```
epochs = 100
learning_rate = 5e-2
batch_size = 128
n_components = 5 # the number of kernel
target_distribution = Normal(0.0, 1.0)
```

[12]:
```
####################################
# STEP 1: Implement 1-d Flow model #
# Model is misture of Gaussian CDFs
####################################

class Flow1d(nn.Module):
    def __init__(self, n_components):
        super(Flow1d, self).__init__()
        self.mus = nn.Parameter(torch.randn(n_components), requires_grad=True)
        self.log_sigmas = nn.Parameter(torch.zeros(n_components), requires_grad=True)
        self.weight_logits = nn.Parameter(torch.ones(n_components), requires_grad=True)

    def forward(self, x):
        x = x.view(-1,1)
        weights = self.weight_logits.exp()
        distribution = Normal(self.mus, self.log_sigmas.exp())
        z = ((distribution.cdf(x) - 0.5) * weights).sum(dim=1)
        dz_by_dx = (distribution.log_prob(x).exp() * weights).sum(dim=1)
        return z, dz_by_dx
```

[13]:
```
################################################
# STEP 2: Create Dataset and Create Dataloader #
################################################

def mixture_of_gaussians(num, mu_var=(-1,0.25, 0.2,0.25, 1.5,0.25)):
    n = num // 3
    m1,s1,m2,s2,m3,s3 = mu_var
    gaussian1 = np.random.normal(loc=m1, scale=s1, size=(n,))
    gaussian2 = np.random.normal(loc=m2, scale=s2, size=(n,))
    gaussian3 = np.random.normal(loc=m3, scale=s3, size=(num-n,))
    return np.concatenate([gaussian1, gaussian2, gaussian3])

class MyDataset(data.Dataset):
    def __init__(self, array):
        super().__init__()
        self.array = array

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

    def __getitem__(self, index):
        return self.array[index]
```

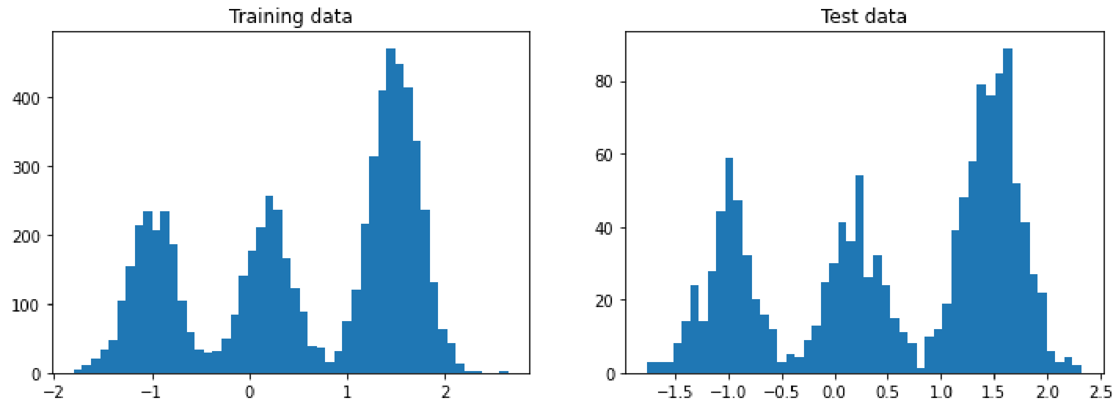
[14]:
```
################################
# STEP 3: Define Loss Function #
################################
```

```
def loss_function(target_distribution, z, dz_by_dx):
    # log(p_Z(z)) = target_distribution.log_prob(z)
    # log(dz/dx) = dz_by_dx.log() (flow is defined so that dz/dx>0)
    log_likelihood = target_distribution.log_prob(z) + dz_by_dx.log()
    return -log_likelihood.mean()  #flip sign, and sum of data X_1,...X_N
```

[15]:
```
###########################
# STEP 4: Train the model #
###########################

# create dataloader
n_train, n_test = 5000, 1000
train_data = mixture_of_gaussians(n_train)
test_data = mixture_of_gaussians(n_test)

train_loader = data.DataLoader(MyDataset(train_data), batch_size=batch_size,␣
 ↪shuffle=True)
test_loader = data.DataLoader(MyDataset(test_data), batch_size=batch_size,␣
 ↪shuffle=True)

# create model
flow = Flow1d(n_components)
optimizer = torch.optim.Adam(flow.parameters(), lr=learning_rate)

train_losses, test_losses = [], []

for epoch in range(epochs):
    # train
#     flow.train()
    mean_loss = 0
    for i, x in enumerate(train_loader):
        z, dz_by_dx = flow(x)
        loss = loss_function(target_distribution, z, dz_by_dx)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        mean_loss += loss.item()
    train_losses.append(mean_loss/(i+1))

    # test
    flow.eval()
    mean_loss = 0
    for i, x in enumerate(test_loader):
        z, dz_by_dx = flow(x)
        loss = loss_function(target_distribution, z, dz_by_dx)

        mean_loss += loss.item()
    test_losses.append(mean_loss/(i+1))
```

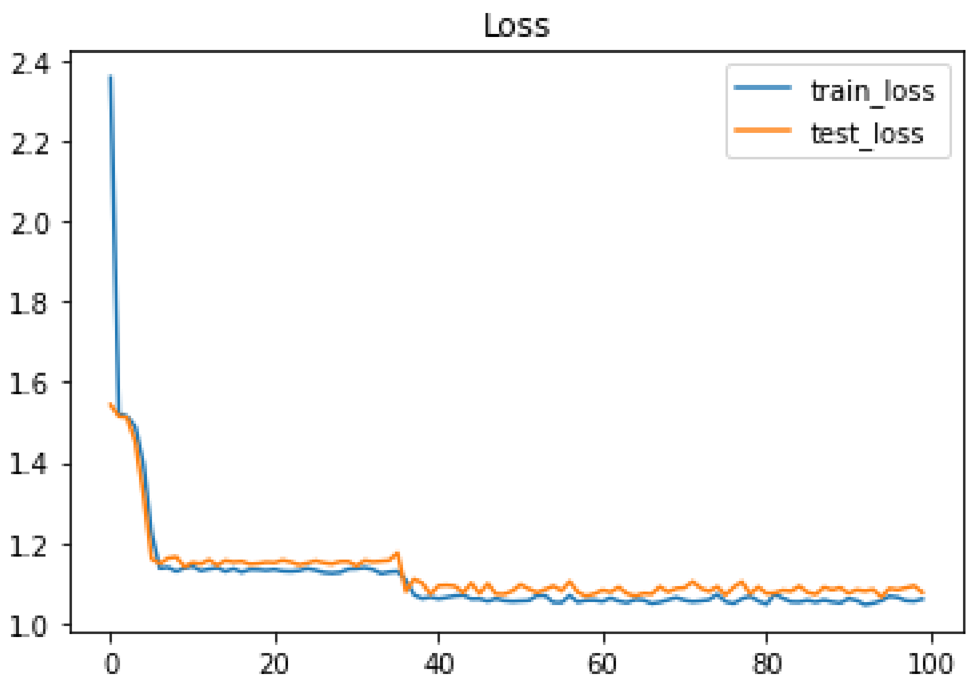
[16]:
```
# Visualize training and test data

_, axes = plt.subplots(1,2, figsize=(12,4))
```

```
_ = axes[0].hist(train_loader.dataset.array, bins=50)
_ = axes[1].hist(test_loader.dataset.array, bins=50)
_ = axes[0].set_title('Training data')
_ = axes[1].set_title('Test data')
```

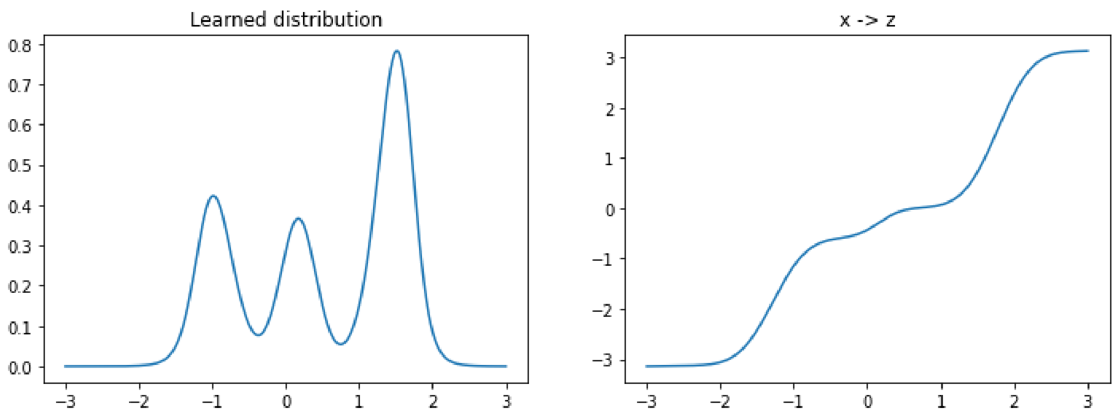


[17]:
```
# View training and test loss

plt.plot(train_losses, label='train_loss')
plt.plot(test_losses, label='test_loss')
plt.title("Loss")
plt.legend()
```

[17]: <matplotlib.legend.Legend at 0x13e5b1040>

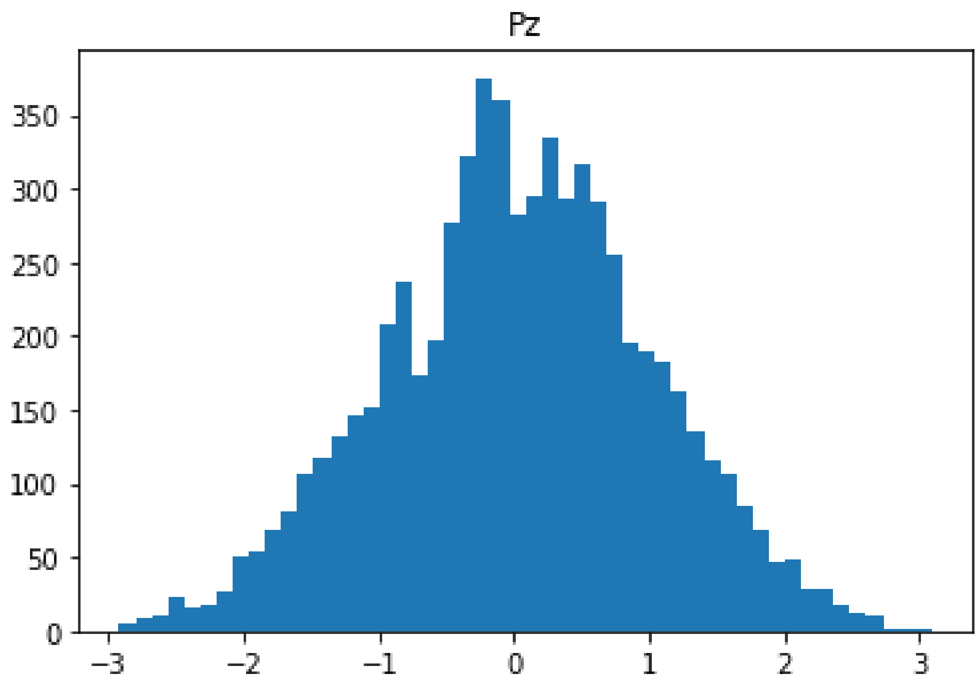


[18]:
```
# View learned distribution and flow map

x = np.linspace(-3,3,1000)
with torch.no_grad():
    z, dz_by_dx = flow(torch.FloatTensor(x))
    px = (target_distribution.log_prob(z) + dz_by_dx.log()).exp().cpu().numpy()

_, axes = plt.subplots(1,2, figsize=(12,4))
_ = axes[0].plot(x,px)
_ = axes[0].set_title('Learned distribution')

_ = axes[1].plot(x,z)
_ = axes[1].set_title('x -> z')
```



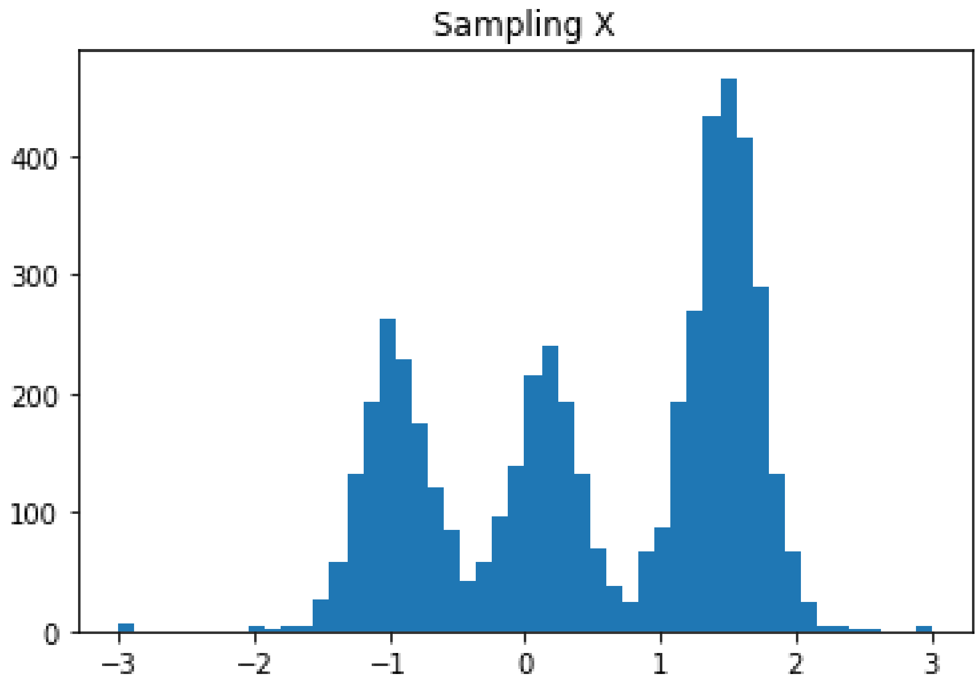
[19]:
```
# View learned pz

with torch.no_grad():
    z, _ = flow(torch.FloatTensor(train_loader.dataset.array))

_ = plt.hist(np.array(z), bins=50)
plt.title("Pz")
```

[19]: Text(0.5, 1.0, 'Pz')



[20]:
```
# Sampling X

N = 5000
z = torch.normal(torch.zeros(N), torch.ones(N))
x_low = torch.full((N,), -3.)
x_high = torch.full((N,), 3.)

#Perform bisection
with torch.no_grad():
    for _ in range(30):
        m = (x_low+x_high)/2
        f,_ = flow(m)
        x_high[f>=z] = m[f>=z]
        x_low[f<z] = m[f<z]
    x = (x_low+x_high)/2

_ = plt.hist(np.array(x), bins=50)
plt.title("Sampling X")
```

[20]: Text(0.5, 1.0, 'Sampling X')


Sampling X
400
300
200
100
0
−3
−2
−1
0
1
2
3