Problem 3

In [ ]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import transforms

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
batch_size = 128
```

In [ ]:
```
test_val_dataset = datasets.MNIST(root='./mnist_data/',
                                train=False,
                                transform=transforms.ToTensor(),
                                download=True)

test_dataset, validation_dataset = ₩
    torch.utils.data.random_split(test_val_dataset, [5000, 5000])

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

In [ ]:
```
# Define prior distribution
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)

# 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_)
    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)))

  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 [ ]:
```
nice = NICE(in_out_dim=784, mid_dim=1000, hidden=5).to(device)
nice.load_state_dict(torch.load('nice.pt', map_location=device))
```

Step 4.

Use validation data to get mean, standard deviation, and threshold.

In [ ]:
```
validation_loader = torch.utils.data.DataLoader(validation_dataset, batch_size=batcl

nice.eval()
likelihood = []
for batch, (data, _) in enumerate(validation_loader):
    data = data.to(device).view(-1, 784)
    l = nice(data)
    likelihood.append(l)

likelihood = torch.cat(likelihood)

mean = likelihood.mean()
std = likelihood.std()

threshold = mean - 3*std
print(f"mean : {mean:.4f}, std : {std:.4f}, threshold : {threshold:.4f}")
```

```
mean : 2255.0303, std : 938.3728, threshold : -560.0881
```

Step 5.

Get type 1 error, by the same way as step 4.

In [ ]:
```
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)

total = 0
anomalies = 0
for batch, (data,_) in enumerate(test_loader):
    data = data.to(device).view(-1, 784)
    l = nice(data)
    detected = torch.where(l < threshold, 1, 0)

    total += data.size()[0]
    anomalies += torch.sum(detected).item()

print(f"{anomalies} anomaly detected among {total} data. ₩nType 1 error : {(anomalie
```

```
91 anomaly detected among 5000 data.
Type 1 error : 0.018
```

Step 6.

Repeat step 5 with KMNIST data to get type 2 error.

In [ ]:
```
k_test_loader = torch.utils.data.DataLoader(anomaly_dataset, batch_size=batch_size)

total = 0
anomalies = 0
for batch, (data,_) in enumerate(k_test_loader):
    data = data.to(device).view(-1, 784)
```

```
    l = nice(data)
    detected = torch.where(l > threshold, 1, 0)

    total += data.size()[0]
    anomalies += torch.sum(detected).item()

print(f"{anomalies} anomaly detected among {total} data. ₩nType 2 error : {(anomalie
```

```
15 anomaly detected among 10000 data.
Type 2 error : 0.002
```