Solution template for the question 1.6-1.7. This template consists of following steps. Except the step 2, you don't need to modify it to answer the questions.

1. Initialize libraries
2. **Insert the answers for the questions 1.1~1.5 below (this is the part you need to fill)**
3. Define data loaders
4. Define VAE network architecture
5. Initialize the model and optimizer
6. Train the model
7. Save the model
8. Load the model
9. Evaluate the model with importance sampling

Initialize libraries

In [22]:

```
import math
from torchvision.datasets import utils
import torch.utils.data as data_utils
import torch
import os
import numpy as np
from torch import nn
from torch.nn.modules import upsampling
from torch.functional import F
from torch.optim import Adam
```

Insert **the answers for the questions 1.1~1.5 below**

In [23]:

```
def log_likelihood_bernoulli(mu, target):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests mi
ght fail.

    *** note. ***

    :param mu: (FloatTensor) - shape: (batch_size x input_size) - The mean of Be
rnoulli random variables p(x=1).
    :param target: (FloatTensor) - shape: (batch_size x input_size) - Target sam
ples (binary values).
    :return: (FloatTensor) - shape: (batch_size,) - log-likelihood of target sam
ples on the Bernoulli random variables.
    """
    # init
    batch_size = mu.size(0)
    mu = mu.view(batch_size, -1)
    target = target.view(batch_size, -1)

    loss = (target*torch.log(mu) + (1-target)*torch.log(1-mu))
    out = torch.sum(loss, dim=-1)
    return out


def log_likelihood_normal(mu, logvar, z):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests mi
ght fail.

    *** note. ***

    :param mu: (FloatTensor) - shape: (batch_size x input_size) - The mean of No
rmal distributions.
    :param logvar: (FloatTensor) - shape: (batch_size x input_size) - The log va
riance of Normal distributions.
    :param z: (FloatTensor) - shape: (batch_size x input_size) - Target samples.
    :return: (FloatTensor) - shape: (batch_size,) - log probability of the sames
on the given Normal distributions.
    """
    # init
    batch_size = mu.size(0)
    mu = mu.view(batch_size, -1)
    logvar = logvar.view(batch_size, -1)
    z = z.view(batch_size, -1)
    out = torch.zeros(batch_size,)
    for i in range(batch_size):
        m = torch.distributions.multivariate_normal.MultivariateNormal(mu[i], to
rch.diag(logvar[i].exp()))
        out[i] = m.log_prob(z[i])
    return out


def log_mean_exp(y):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests mi
ght fail.

    *** note. ***
```

```
    :param y: (FloatTensor) - shape: (batch_size x sample_size) - Values to be e
valuated for log_mean_exp. For example log proabailies
    :return: (FloatTensor) - shape: (batch_size,) - Output for log_mean_exp.
    """
    # init
    batch_size = y.size(0)
    sample_size = y.size(1)
    out = torch.zeros(batch_size,)

    for i in range(batch_size):
        yi_max = torch.max(y[i])
        yi = y[i] - yi_max
        out[i] = torch.log(torch.mean(torch.exp(yi))) + yi_max

    return out


def kl_gaussian_gaussian_analytic(mu_q, logvar_q, mu_p, logvar_p):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests mi
ght fail.

    *** note. ***

    :param mu_q: (FloatTensor) - shape: (batch_size x input_size) - The mean of
 first distributions (Normal distributions).
    :param logvar_q: (FloatTensor) - shape: (batch_size x input_size) - The log
 variance of first distributions (Normal distributions).
    :param mu_p: (FloatTensor) - shape: (batch_size x input_size) - The mean of
 second distributions (Normal distributions).
    :param logvar_p: (FloatTensor) - shape: (batch_size x input_size) - The log
 variance of second distributions (Normal distributions).
    :return: (FloatTensor) - shape: (batch_size,) - kl-divergence of KL(q||p).
    """
    # init
    batch_size = mu_q.size(0)
    mu_q = mu_q.view(batch_size, -1)
    logvar_q = logvar_q.view(batch_size, -1)
    mu_p = mu_p.view(batch_size, -1)
    logvar_p = logvar_p.view(batch_size, -1)
    out = torch.ones(batch_size,).type(torch.FloatTensor)

    for i in range(batch_size):
        sigma0 = torch.diag(logvar_q[i].exp())
        sigma1 = torch.diag(logvar_p[i].exp())
        mu0 = mu_q[i]
        mu1 = mu_p[i]

        ## find KL(N0||N1)
        ## https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence#Mul
tivariate_normal_distributions
        ## out[i] = (torch.trace(torch.matmul(torch.inverse(sigma1), sigma0)) +
 torch.matmul((mu1 - mu0), torch.matmul(torch.inverse(sigma1), (mu1 - mu0))) - l
en(mu1) + torch.log(torch.det(sigma1)/torch.det(sigma0)))/2.0

        dq = torch.distributions.multivariate_normal.MultivariateNormal(mu_q[i],
torch.diag(logvar_q[i].exp()))
        dp = torch.distributions.multivariate_normal.MultivariateNormal(mu_p[i],
torch.diag(logvar_p[i].exp()))
        out[i] = torch.distributions.kl.kl_divergence(dq, dp)
```

```python
    return out


def kl_gaussian_gaussian_mc(mu_q, logvar_q, mu_p, logvar_p, num_samples=1):
    """
    COMPLETE ME. DONT MODIFY THE PARAMETERS OF THE FUNCTION. Otherwise, tests mi
ght fail.

    *** note. ***

    :param mu_q: (FloatTensor) - shape: (batch_size x input_size) - The mean of
 first distributions (Normal distributions).
    :param logvar_q: (FloatTensor) - shape: (batch_size x input_size) - The log
 variance of first distributions (Normal distributions).
    :param mu_p: (FloatTensor) - shape: (batch_size x input_size) - The mean of
 second distributions (Normal distributions).
    :param logvar_p: (FloatTensor) - shape: (batch_size x input_size) - The log
 variance of second distributions (Normal distributions).
    :param num_samples: (int) - shape: () - The number of sample for Monte Carlo
estimate for KL-divergence
    :return: (FloatTensor) - shape: (batch_size,) - kl-divergence of KL(q||p).
    """
    # init

    batch_size = mu_q.size(0)
    input_size = np.prod(mu_q.size()[1:])
    out = torch.ones(batch_size,)
    out = out.type(torch.FloatTensor)


    for i in range(batch_size):
        dq = torch.distributions.multivariate_normal.MultivariateNormal(mu_q[i],
torch.diag(logvar_q[i].exp()))
        dp = torch.distributions.multivariate_normal.MultivariateNormal(mu_p[i],
torch.diag(logvar_p[i].exp()))
        kl_div = 0
        for j in range(num_samples):
            sample = dq.sample()
            kl_div += dq.log_prob(sample) - dp.log_prob(sample)

        out[i] = kl_div / float(num_samples)

    return out
```

Define data loaders

In [24]:

```
def get_data_loader(dataset_location, batch_size):
    URL = "http://www.cs.toronto.edu/~larocheh/public/datasets/binarized_mnist/"
    # start processing
    def lines_to_np_array(lines):
        return np.array([[int(i) for i in line.split()] for line in lines])
    splitdata = []
    for splitname in ["train", "valid", "test"]:
        filename = "binarized_mnist_%s.amat" % splitname
        filepath = os.path.join(dataset_location, filename)
        utils.download_url(URL + filename, dataset_location)
        with open(filepath) as f:
            lines = f.readlines()
        x = lines_to_np_array(lines).astype('float32')
        x = x.reshape(x.shape[0], 1, 28, 28)
        # pytorch data loader
        dataset = data_utils.TensorDataset(torch.from_numpy(x))
        dataset_loader = data_utils.DataLoader(x, batch_size=batch_size, shuffle
=splitname == "train")
        splitdata.append(dataset_loader)
    return splitdata
```

In [25]:

```
train, valid, test = get_data_loader("binarized_mnist", 64)
```

```
Using downloaded and verified file: binarized_mnist/binarized_mnist_
train.amat
Using downloaded and verified file: binarized_mnist/binarized_mnist_
valid.amat
Using downloaded and verified file: binarized_mnist/binarized_mnist_
test.amat
```

Define VAE network architecture

In [26]:

```
class Encoder(nn.Module):
    def __init__(self, latent_size):
        super(Encoder, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(784, 300),
            nn.ELU(),
            nn.Linear(300, 300),
            nn.ELU(),
            nn.Linear(300, 2 * latent_size),
        )

    def forward(self, x):
        batch_size = x.size(0)
        z_mean, z_logvar = self.mlp(x.view(batch_size, 784)).chunk(2, dim=-1)
        return z_mean, z_logvar

class Decoder(nn.Module):
    def __init__(self, latent_size):
        super(Decoder, self).__init__()
        self.mlp = nn.Sequential(
            nn.Linear(latent_size, 300),
            nn.ELU(),
            nn.Linear(300, 300),
            nn.ELU(),
            nn.Linear(300, 784),
        )

    def forward(self, z):
        return self.mlp(z) - 5.

class VAE(nn.Module):
    def __init__(self, latent_size):
        super(VAE, self).__init__()
        self.encode = Encoder(latent_size)
        self.decode = Decoder(latent_size)

    def forward(self, x):
        z_mean, z_logvar = self.encode(x)
        z_sample = z_mean + torch.exp(z_logvar / 2.) * torch.randn_like(z_logvar
)
        x_mean = self.decode(z_sample)
        return z_mean, z_logvar, x_mean

    def loss(self, x, z_mean, z_logvar, x_mean):
        ZERO = torch.zeros(z_mean.size())
        #kl = kl_gaussian_gaussian_mc(z_mean, z_logvar, ZERO, ZERO, num_samples=
1000).mean()
        kl = kl_gaussian_gaussian_analytic(z_mean, z_logvar, ZERO, ZERO).mean()
        recon_loss = -log_likelihood_bernoulli(
            torch.sigmoid(x_mean.view(x.size(0), -1)),
            x.view(x.size(0), -1),
        ).mean()
        return recon_loss + kl
```

Initialize a model and optimizer

In [27]:

```
vae = VAE(100)
params = vae.parameters()
optimizer = Adam(params, lr=3e-4)
print(vae)
```

```
VAE(
  (encode): Encoder(
    (mlp): Sequential(
      (0): Linear(in_features=784, out_features=300, bias=True)
      (1): ELU(alpha=1.0)
      (2): Linear(in_features=300, out_features=300, bias=True)
      (3): ELU(alpha=1.0)
      (4): Linear(in_features=300, out_features=200, bias=True)
    )
  )
  (decode): Decoder(
    (mlp): Sequential(
      (0): Linear(in_features=100, out_features=300, bias=True)
      (1): ELU(alpha=1.0)
      (2): Linear(in_features=300, out_features=300, bias=True)
      (3): ELU(alpha=1.0)
      (4): Linear(in_features=300, out_features=784, bias=True)
    )
  )
)
```

Train the model

In [28]:

```
for i in range(20):
    # train
    for x in train:
        optimizer.zero_grad()
        z_mean, z_logvar, x_mean = vae(x)
        loss = vae.loss(x, z_mean, z_logvar, x_mean)
        loss.backward()
        optimizer.step()

    # evaluate ELBO on the valid dataset
    with torch.no_grad():
        total_loss = 0.
        total_count = 0
        for x in valid:
            total_loss += vae.loss(x, *vae(x)) * x.size(0)
            total_count += x.size(0)
        print('-elbo: ', (total_loss / total_count).item())
```

```
-elbo:  167.1219940185547
-elbo:  142.9504852294922
-elbo:  129.03280639648438
-elbo:  122.30564880371094
-elbo:  116.72787475585938
-elbo:  113.34302520751953
-elbo:  110.86123657226562
-elbo:  108.96890258789062
-elbo:  107.67586517333984
-elbo:  106.59246063232422
-elbo:  105.75444793701172
-elbo:  104.75553131103516
-elbo:  104.35298919677734
-elbo:  104.01703643798828
-elbo:  103.03252410888672
-elbo:  102.75845336914062
-elbo:  102.19798278808594
-elbo:  102.01860046386719
-elbo:  101.64897155761719
-elbo:  101.37193298339844
```

Save the model

In [29]:

```
torch.save(vae, 'model.pt')
```

```
/home/dhaivat1729/anaconda3/envs/detectron_fair/lib/python3.6/site-p
ackages/torch/serialization.py:292: UserWarning: Couldn't retrieve s
ource code for container of type VAE. It won't be checked for correc
tness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/home/dhaivat1729/anaconda3/envs/detectron_fair/lib/python3.6/site-p
ackages/torch/serialization.py:292: UserWarning: Couldn't retrieve s
ource code for container of type Encoder. It won't be checked for co
rrectness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/home/dhaivat1729/anaconda3/envs/detectron_fair/lib/python3.6/site-p
ackages/torch/serialization.py:292: UserWarning: Couldn't retrieve s
ource code for container of type Sequential. It won't be checked for
correctness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/home/dhaivat1729/anaconda3/envs/detectron_fair/lib/python3.6/site-p
ackages/torch/serialization.py:292: UserWarning: Couldn't retrieve s
ource code for container of type Linear. It won't be checked for cor
rectness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/home/dhaivat1729/anaconda3/envs/detectron_fair/lib/python3.6/site-p
ackages/torch/serialization.py:292: UserWarning: Couldn't retrieve s
ource code for container of type ELU. It won't be checked for correc
tness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
/home/dhaivat1729/anaconda3/envs/detectron_fair/lib/python3.6/site-p
ackages/torch/serialization.py:292: UserWarning: Couldn't retrieve s
ource code for container of type Decoder. It won't be checked for co
rrectness upon loading.
  "type " + obj.__name__ + ". It won't be checked "
```

Load the model

In [30]:

```
vae = torch.load('model.pt')
```

Evaluate the $\log p_\theta(x)$ of the model on test by using importance sampling

In [31]:

```
total_loss = 0.
total_count = 0
with torch.no_grad():
    #x = next(iter(test))
    for x in test:
        # init
        K = 200
        M = x.size(0)

        # Sample from the posterior
        z_mean, z_logvar = vae.encode(x)
        eps = torch.randn(z_mean.size(0), K, z_mean.size(1))
        z_samples = z_mean[:, None, :] + torch.exp(z_logvar / 2.)[:, None, :] * 
eps # Broadcast the noise over the mean and variance

        # Decode samples
        z_samples_flat = z_samples.view(-1, z_samples.size(-1)) # Flatten out th
e z samples
        x_mean_flat = vae.decode(z_samples_flat) # Push it through

        # Reshape images and posterior to evaluate probabilities
        x_flat = x[:, None].repeat(1, K, 1, 1, 1).reshape(M*K, -1)
        z_mean_flat = z_mean[:, None, :].expand_as(z_samples).reshape(M*K, -1)
        z_logvar_flat =  z_logvar[:, None, :].expand_as(z_samples).reshape(M*K, 
-1)
        ZEROS = torch.zeros(z_mean_flat.size())

        # Calculate all the probabilities!
        log_p_x_z = log_likelihood_bernoulli(torch.sigmoid(x_mean_flat), x_flat)
.view(M, K)
        log_q_z_x = log_likelihood_normal(z_mean_flat, z_logvar_flat, z_samples_
flat).view(M, K)
        log_p_z = log_likelihood_normal(ZEROS, ZEROS, z_samples_flat).view(M, K)

        # Recombine them.
        w = log_p_x_z + log_p_z - log_q_z_x
        log_p = log_mean_exp(w)

        # Accumulate
        total_loss += log_p.sum()
        total_count += M

print('log p(x):', (total_loss / total_count).item())
```

```
log p(x): -95.58343505859375
```

In [ ]:

```

```