Variational Auto-Encoders

October 28, 2022

1 Variational inference

- Representation of the data though a **latent variable z**:

$$p(\mathbf{x}) = \int_{\mathbf{z}} p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}.$$

- **Neural network** to approximate $p(\mathbf{x}|\mathbf{z};\theta)$.
- When evaluating the likelihood on our data, we need samples from $\mathbf{z}$ that will give high values of $p(\mathbf{x}|\mathbf{z})$.
- Instead of computing the distribution, or approximating the real posterior $p(\mathbf{z}|\mathbf{x})$ by sampling from it, we choose an **approximated posterior distribution on z** and try to make it resemble the real posterior as close as possible.

We use a **family** of distributions, parameterized by some parameters ϕ (e.g. they can be, again, the **parameters of a neural network**)

$$q(\mathbf{z};\phi).$$

Variational inference consists in optimizing ϕ **simultaneously** with θ so that $q(\mathbf{z};\phi)$ is as close as possible to $p(\mathbf{z}|\mathbf{x};\theta)$ and maximizing a lower bound on the log-likelihood.

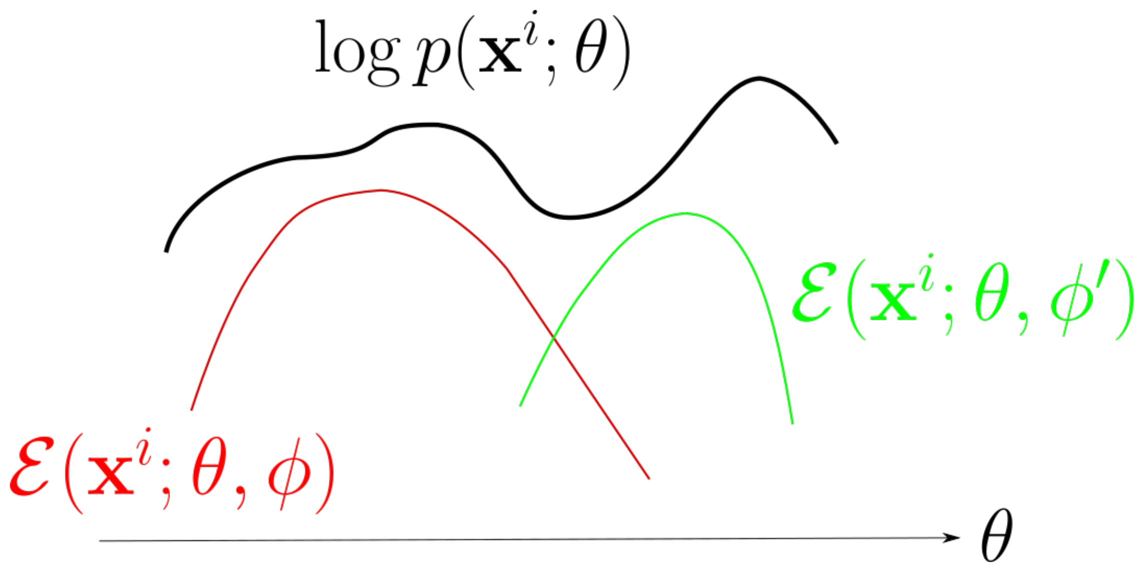
For one training data $\mathbf{x}^i$

$$\mathcal{L}(\mathbf{x}^i;\theta) \triangleq \log p(\mathbf{x}^i;\theta) \geq \int_{\mathbf{z}} q(\mathbf{z};\phi)\log p(\mathbf{x}^i,\mathbf{z};\theta)d\mathbf{z} + H(q(\mathbf{z};\phi)) \triangleq \mathcal{E}(\mathbf{x}^i;\theta,\phi) \text{ (ELBO)}$$

and

$$\mathcal{L}(\mathbf{x}^i;\theta) = \mathcal{E}(\mathbf{x}^i;\theta,\phi) + D_{KL}(q(\mathbf{z};\phi)\|p(\mathbf{z}|\mathbf{x}^i;\theta))$$

As $q(\mathbf{z};\phi)$ approximate well the posterior $p(\mathbf{z}|\mathbf{x}^i;\theta)$, the lower bound $\mathcal{E}(\mathbf{x}^i;\theta,\phi)$ will be close to $\log p(\mathbf{x}^i;\theta)$, which means that **maximizing the ELBO will be very similar to maximizing the marginal likelihood**. This process is called **variational inference**.



Training a model implies maximizing its log-likelihood over a dataset $\mathcal{D}_{train}$

$$\mathcal{L}(\mathcal{D};\theta) = \sum_{\mathbf{x}_i \in \mathcal{D}} \log p(\mathbf{x}_i;\theta).$$

Evidence lower bound (ELBO) holds for **any** $q(\mathbf{z};\phi)$

$$\log p(\mathbf{x};\theta) \geq \sum_{\mathbf{z}\sim q(\mathbf{z};\phi)} \log p(\mathbf{z},\mathbf{x};\theta) + H(q(\mathbf{z};\phi)) = \mathcal{E}(\mathbf{x};\theta,\phi).$$

To train the model, we maximize the total likelihood over $\mathcal{D}_{train}$:

$$\mathcal{L}(\mathcal{D}_{train};\theta) = \sum_{\mathbf{x}^i \in \mathcal{D}_{train}} \log p(\mathbf{x}^i;\theta) \geq \sum_{\mathbf{x}^i \in \mathcal{D}_{train}} \mathcal{E}(\mathbf{x}^i;\theta,\phi^i)$$

Hence,

$$\max_{\theta} \mathcal{L}(\mathcal{D}_{train};\theta) \geq \max_{\theta,\phi^1,\ldots,\phi^n} \sum_{\mathbf{x}^i \in \mathcal{D}_{train}} \mathcal{E}(\mathbf{x}^i;\theta,\phi^i).$$

Note that we use different variational parameters ϕ^i for every data point $\mathbf{x}^i$, because the true posterior $p(\mathbf{z}|\mathbf{x}^i;\theta)$ **should be different from one data $\mathbf{x}^i$ to another**.

To evaluate the bound $\mathcal{E}(\mathbf{x}^i;\theta,\phi^i)$, sample $\mathbf{z}^{(1)}, \ldots, \mathbf{z}^{(S)}$ from $q(\mathbf{z};\phi^i)$ and estimate

$$\mathcal{E}(\mathbf{x}^i;\theta,\phi^i) = E_{q(\mathbf{z};\phi^i)}[\log p(\mathbf{z},\mathbf{x}^i;\theta) - \log q(\mathbf{z};\phi^i)] \approx \frac{1}{S}\sum_{s=1}^{S} \log p(\mathbf{z}^{(s)},\mathbf{x}^i;\theta) - \log q(\mathbf{z}^{(s)};\phi))$$

Note that for this evaluation to be implemented, we need p and q to be **evaluated and sampled**: simple distributions (**Gaussians**, for example).

To **maximize** this bound, a general strategy is to follow the stochastic gradient algorithm and, for each data $\mathbf{x}^i$, **alternate the optimization of ϕ^i and the one of θ.**

Stochastic variational inference

1. Initialize $\theta, \phi^1, ..., \phi^n$.
2. **Sample a data point $\mathbf{x}^i$** from $\mathcal{D}_{train}$.
3. Optimize $\mathcal{E}(\mathbf{x}^i; \theta, \phi^i)$ with respect to ϕ^i: determine $\phi^{i*} \approx \arg\max_{\phi^i} \mathcal{E}(\mathbf{x}^i; \theta, \phi^i)$ by **gradient ascent steps**:

$$\phi^i = \phi^i + \alpha \nabla_{\phi^i} \mathcal{E}(\mathbf{x}^i; \theta; \phi^i).$$

4. Compute $\nabla_\theta \mathcal{E}(\mathbf{x}^i; \theta, \phi^{i*})$.
5. Update θ **in the gradient direction**:

$$\theta = \theta + \beta \nabla_\theta \mathcal{E}(\mathbf{x}^i; \theta; \phi^i).$$

6. Follow on step 2.

The gradient with respect to θ is simple because **only the first term in the ELBO depends on θ**:

$$\begin{aligned} \nabla_\theta E_{q(\mathbf{z};\phi)}[\log p(\mathbf{z}, \mathbf{x}; \theta) - \log q(\mathbf{z}; \phi)] = & \quad E_{q(\mathbf{z};\phi)}[\nabla_\theta \log p(\mathbf{z}, \mathbf{x}; \theta)], & (1) \\ \approx & \quad \frac{1}{S} \sum_{s=1}^{S} \nabla_\theta \log p(\mathbf{z}^{(s)}, \mathbf{x}; \theta). & (2) \end{aligned}$$

Now, computing

$$\nabla_\phi E_{q(\mathbf{z};\phi)}[\log p(\mathbf{z}, \mathbf{x}; \theta) - \log q(\mathbf{z}; \phi)]$$

is a prori **way more complicated** because what we have is an **expectation over a distribution parameterized through ϕ**. How to take the derivative of this kind of expression on ϕ?

1.1 The reparameterization trick

Consider, more generally, the expectation of a scalar function f:

$$E_{q(\mathbf{z};\phi)}[f(\mathbf{z}; \phi)] = \int f(\mathbf{z}; \phi) q(\mathbf{z}; \phi) d\mathbf{z}$$

and suppose that q is simply a **Gaussian with diagonal covariance**:

$$q(\mathbf{z}; \phi) \triangleq \mathcal{N}(\mu, \sigma^2 \mathbf{I}).$$

Sampling $\mathbf{z}$ from q can be written as:

$$\epsilon \sim \mathcal{N}(0, \mathbf{I}) \text{ and } \mathbf{z} = \mu(\phi) + \sigma(\phi)\epsilon.$$

Then

$$E_{q(\mathbf{z};\phi)}[f(\mathbf{z};\phi)] = \int_{\mathbf{z}\sim q(\mathbf{z};\phi)} f(\mathbf{z};\phi)q(\mathbf{z};\phi)d\mathbf{z} = \int_{\epsilon} f(\mu(\phi) + \sigma(\phi)\epsilon;\phi)p(\epsilon)d\epsilon.$$

This way, we **remove the ϕ from the integral**! This works more generally whenever you can express

$$\mathbf{z} \sim q(\mathbf{z};\phi) \text{ as } \mathbf{z} = g(\epsilon;\phi).$$

Then:

$$\begin{aligned} \nabla_\phi E_{q(\mathbf{z};\phi)}[f(\mathbf{z};\phi)] = & \quad \nabla_\phi \int_\epsilon f(\mu(\phi) + \sigma(\phi)\epsilon;\phi)p(\epsilon)d\epsilon && (3) \\ = & \quad \int_\epsilon \nabla_\phi f(\mu(\phi) + \sigma(\phi)\epsilon;\phi)d\epsilon. && (4) \end{aligned}$$

From this we can use **Monte-Carlo again**:

$$\nabla_\phi E_{q(\mathbf{z};\phi)}[f(\mathbf{z};\phi)] \approx \frac{1}{S}\sum_{s=1}^{S} \nabla_\phi f(\mu(\phi) + \sigma(\phi)\epsilon^{(s)};\phi),$$

where $\epsilon^{(1)}, \epsilon^{(2)}, ..., \epsilon^{(S)}$ are sampled from $\mathcal{N}(0, \mathbf{I})$. The last expression should be differentiable if f, μ, σ are differentiable.

Translated in our problem, for a given data point $\mathbf{x}^i$:

$$\nabla_{\phi^i} E_{q(\mathbf{z};\phi^i)}[\log p(\mathbf{z}, \mathbf{x}^i;\theta) - \log q(\mathbf{z};\phi^i)] \approx \frac{1}{S}\sum_{s=1}^{S}\left[\nabla_{\phi^i} \log p(\mu(\phi^i) + \sigma(\phi^i)\epsilon^{(s)}, \mathbf{x}^i;\theta) - \log q(\mu(\phi^i) + \sigma(\phi^i)\epsilon^{(s)})\right].$$

However, this plan of estimating one ϕ^i per data is hardly scalable!

1.2 Amortized inference

Instead of learning one function per data point, what about learning **only one function** that would **depend on the data point $\mathbf{x}^i$** and would change the characteristics of the p.d.f. in function of the data point:

$$\mathbf{x}^i \to q(\mathbf{z}|\mathbf{x}^i;\phi) \approx \phi^{i*}.$$

That would replace the step of optimizing ϕ^i for each data, by learning a function that estimates this optimal value.

1. Initialize θ, ϕ.
2. **Sample a data point** $\mathbf{x}^i$ from $\mathcal{D}_{train}$.
3. Optimize $\mathcal{E}(\mathbf{x}^i; \theta, \phi)$ with respect to ϕ: update ϕ by **gradient ascent steps** (with the **reparameterization trick!**):

$$\phi = \phi + \alpha \nabla_\phi \mathcal{E}(\mathbf{x}^i; \theta; \phi).$$

4. Compute $\nabla_\theta \mathcal{E}(\mathbf{x}^i; \theta, \phi)$.
5. Update θ **in the gradient direction**:

$$\theta = \theta + \beta \nabla_\theta \mathcal{E}(\mathbf{x}^i; \theta; \phi).$$

6. Follow on step 2.

1.3 The variational auto-encoder

If we rewrite the ELBO:

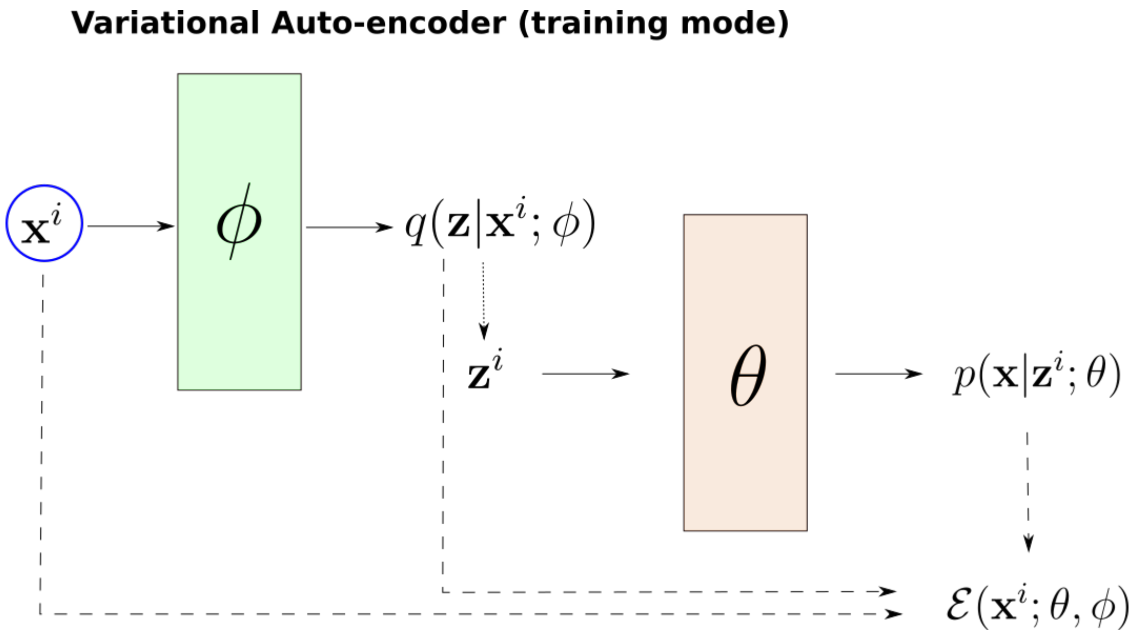
$$\begin{aligned}
\mathcal{E}(\mathbf{x}^i; \theta, \phi^i) &= E_{q(\mathbf{z};\phi)}[\log p(\mathbf{z}, \mathbf{x}^i; \theta) - \log q(\mathbf{z}|\mathbf{x}; \phi)] && (5)\\
&= E_{q(\mathbf{z};\phi)}[\log p(\mathbf{x}^i|\mathbf{z}; \theta) + \log p(\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x}; \phi)] && (6)\\
&= E_{q(\mathbf{z};\phi)}[\log p(\mathbf{x}^i|\mathbf{z}; \theta)] + E_{q(\mathbf{z};\phi)}[\log p(\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x}; \phi)] && (7)\\
&= E_{q(\mathbf{z};\phi)}[\log p(\mathbf{x}^i|\mathbf{z}; \theta)] + D_{KL}(q(\mathbf{z}|\mathbf{x}; \phi), p(\mathbf{z})), && (8)
\end{aligned}$$

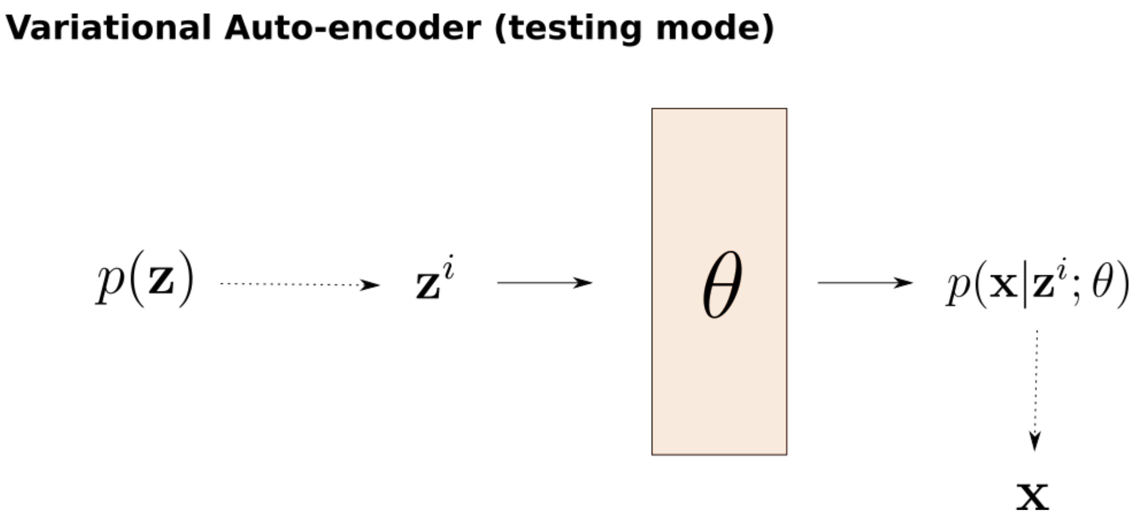
we can re-interpret the objective as:

- a **reconstruction term** $\log p(\mathbf{x}^i|\mathbf{z}; \theta)$ that will favor using sampled $\mathbf{z}$ that produce an $\mathbf{x}$ similar to $\mathbf{x}^i$;
- a second term that favors sampled $\mathbf{z}$ that are likely according to the **prior** $p(\mathbf{z})$.

Encoder-decoder structure: during **training**, for a new data $\mathbf{x}^i$

- compute the **posterior** on $\mathbf{z}$ from $q(\mathbf{z}|\mathbf{x}^i; \phi)$ (**encoder**);
- **sample a latent variable value** $\mathbf{z}^i$ from this posterior;
- deduce $p(\mathbf{x}|\mathbf{z}^i; \theta)$ (**decoder**).





1.4 A useful particular case: Gaussian distributions

Suppose that we limit ourselves to: $z \in \mathbb{R}^d$

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{9}$$

$$q(\mathbf{z}|\mathbf{x};\phi) = \mathcal{N}(\mu_q(\mathbf{x};\phi), diag(\sigma_q(\mathbf{x};\phi))) \tag{10}$$

$$\tag{11}$$

In that case,

$$\mathcal{E}(\mathbf{x}^i;\theta,\phi^i) = E_{q(\mathbf{z};\phi)}[\log p(\mathbf{x}^i|\mathbf{z};\theta)] + D_{KL}(q(\mathbf{z}|\mathbf{x};\phi), p(\mathbf{z})),$$

and the D_{KL} has a **simple analytic form**.

$$D_{KL}(q,p) = -\int q(\mathbf{x}) \log p(\mathbf{x}) d\mathbf{x} + \int q(\mathbf{x}) \log q(\mathbf{x}) \mathbf{d}x$$

First note that in 1D

$$\int q(\mathbf{x}) \log q(\mathbf{x}) d\mathbf{x} = \frac{1}{\sqrt{2\pi\sigma_q^2}} \int [-\frac{1}{2\sigma_q^2}(\mathbf{x}-\mu_q)^2 - \frac{1}{2}\log(2\pi\sigma_q^2)] \exp(-\frac{1}{2\sigma_q^2}(\mathbf{x} - \mu \tag{12}$$

$$= -\frac{1}{2}\log(2\pi\sigma_q^2) \int \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp(-\frac{1}{2\sigma_q^2}(\mathbf{x}-\mu_q)^2) d\mathbf{x} - \frac{1}{2\sigma_q^2} \int (\mathbf{x}-\mu_q)^2 \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp(-\frac{1}{2\sigma_q^2}(\mathbf{x} - \mu \tag{13}$$

$$= -\frac{1}{2}\log(2\pi\sigma_q^2) - \tag{14}$$

$$= -\frac{1}{2}(1 + \log(\tag{15}$$

And

$$\int q(\mathbf{x}) \log p(\mathbf{x}) d\mathbf{x} = \int [-\frac{1}{2\sigma_p^2}(\mathbf{x}-\mu_p)^2 - \frac{1}{2}\log(2\pi\sigma_p^2)] q(\mathbf{x}) d\mathbf{x} \tag{16}$$

$$= -\frac{1}{2}\log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}\int (\mathbf{x}^2 - 2\mu_p\mathbf{x} + \mu_p^2) q(\mathbf{x}) d\mathbf{x} \tag{17}$$

$$= -\frac{1}{2}\log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}\int \mathbf{x}^2 q(\mathbf{x}) d\mathbf{x} + 2\frac{1}{2\sigma_p^2}\int \mu_p \mathbf{x} q(\mathbf{x}) d\mathbf{x} - \frac{1}{2\sigma_p^2}\int \mu_p^2 q(\mathbf{x}) d\mathbf{x} \tag{18}$$

$$= -\frac{1}{2}\log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}(\sigma_q^2 + \mu_q^2) + \frac{1}{\sigma_p^2}\mu_q\mu_p - \frac{1}{2\sigma_p^2}\mu_p^2 \tag{19}$$

$$= -\frac{1}{2}\log(2\pi\sigma_p^2) - \frac{1}{2\sigma_p^2}(\sigma_q^2 + (\mu_q - \mu_p)^2) \tag{20}$$

Then:

$$
\begin{aligned}
D_{KL}(q,p) = & \quad -\int q(\mathbf{x}) \log p(\mathbf{x}) d\mathbf{x} + \int q(\mathbf{x}) \log q(\mathbf{x}) \mathbf{d}x & (21)\\
= & \quad \frac{1}{2}\log(2\pi\sigma_p^2) + \frac{1}{2\sigma_p^2}(\sigma_q^2 + (\mu_q - \mu_p)^2) - \frac{1}{2}(1 + \log(2\pi\sigma_q^2)) & (22)\\
= & \quad \log(\frac{\sigma_p}{\sigma_q}) + \frac{1}{2}\frac{\sigma_q^2}{\sigma_p^2} + \frac{1}{2\sigma_p^2}(\mu_q - \mu_p)^2 - \frac{1}{2}. & (23)
\end{aligned}
$$

When $\mu_p = 0$ and $\sigma_p = 1$, we get

$$
D_{KL}(q,p) = -\log(\sigma_q) + \frac{1}{2}\sigma_q^2 + \frac{1}{2}(\mu_q)^2 - \frac{1}{2},
$$

hence, in our problem (where we can decouple each dimension j)

$$
D_{KL}(q(\mathbf{z}|\mathbf{x};\phi), p(\mathbf{z})) = -\frac{1}{2}\sum_{j=1}^{d}\log(\sigma_{q,j}^2(\mathbf{x};\phi)) + \frac{1}{2}\sum_{j=1}^{d}\sigma_{q,j}^2(\mathbf{x};\phi) + \frac{1}{2}\sum_{j=1}^{d}\mu_{q,j}^2(\mathbf{x};\phi) - \frac{d}{2}.
$$

This expression can be **differentiated easily**.

For ensuring that $\sigma_{q,j} > 0$ during the optimization process, the network outputs instead:

$$
logvar_j(\mathbf{x};\phi) \triangleq \log(\sigma_{q,j}^2(\mathbf{x};\phi))
$$

and the KL becomes:

$$
D_{KL}(q(\mathbf{z}|\mathbf{x};\phi), p(\mathbf{z})) = -\frac{1}{2}\sum_{j=1}^{d} logvar_j(\mathbf{x};\phi) + \frac{1}{2}\sum_{j=1}^{d}\exp(logvar_j(\mathbf{x};\phi)) + \frac{1}{2}\sum_{j=1}^{d}\mu_{q,j}^2(\mathbf{x};\phi) - \frac{d}{2}.
$$

1.5 Example

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
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
import numpy as np
import random
import tqdm
```

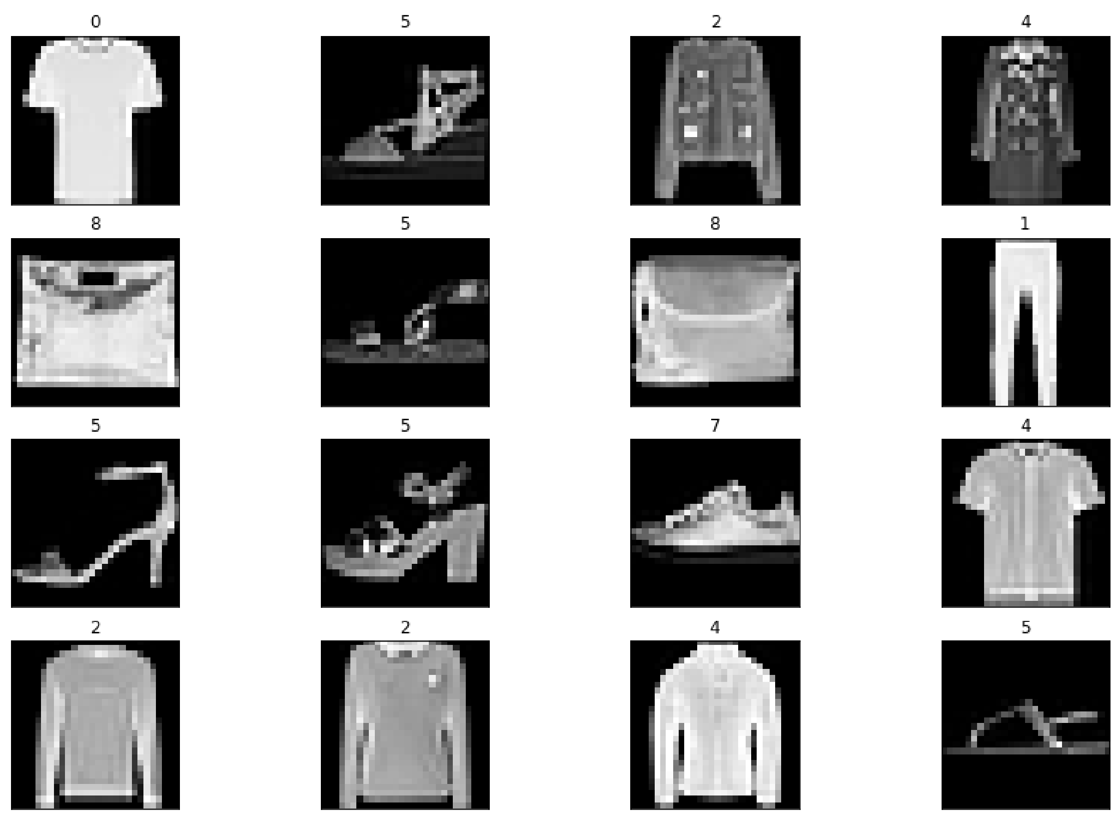
[2]:
```python
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
device
```

[2]: device(type='cuda')

[3]:
```python
# Load the csv files
test_data  = pd.read_csv('data/fashion-mnist_test.csv').values
train_data = pd.read_csv('data/fashion-mnist_train.csv').values
```

[4]:
```python
# Get the normalized images (as vectors) and the labels
x_train = train_data[:,1:]/255
y_train = train_data[:,0]
x_test  = test_data[:,1:]/255
y_test  = test_data[:,0]
```

[5]:
```python
# Sanity check: random samples from the training data
indices = np.random.randint(len(y_train), size=16)
plt.figure(figsize=(15, 10))
for i in range(16):
    plt.subplot(4, 4, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.imshow(x_train[indices[i]].reshape((28,28)),cmap='gray')
    plt.title(y_train[indices[i]])
```



[12]:

```
# Our VAE class
class VAE(nn.Module):
    nFeatures = 64
    imgWidth  = 28
    imgHeight = 28
    dimLatent = 64
    # Constructor
    def __init__(self, dimChannels=1,␣
↪dimFeatures=nFeatures*(imgWidth-6)*(imgHeight-6), dimLatent=dimLatent):
        super(VAE, self).__init__()
        self.dimFeatures = dimFeatures
        self.dimLatent   = dimLatent
        self.dimChannels = dimChannels
        # 16 filters, 5x5
        self.enConv1 = nn.Conv2d(self.dimChannels, 16, 5)
        # 32 filters, 3x3
        self.enConv2 = nn.Conv2d(16, self.nFeatures, 3)
        # Fully connected layers to produce the posterior parameters
        self.enFC1   = nn.Linear(self.dimFeatures, dimLatent)
        self.enFC2   = nn.Linear(self.dimFeatures, dimLatent)
        # Decoder
        self.deFC1   = nn.Linear(dimLatent, dimFeatures)
        self.deConv1 = nn.ConvTranspose2d(self.nFeatures, 16, 3)
        self.deConv2 = nn.ConvTranspose2d(16, dimChannels, 5)

    # Encoding: produce q(z|x;phi)
    def encode(self, x):
        x = F.relu(self.enConv1(x))
        x = F.relu(self.enConv2(x))
        x = x.view(-1,self.dimFeatures)
        # Produce q(z|x;phi)
        mu        = self.enFC1(x)
        logsigma  = self.enFC2(x)
        return mu, logsigma

    # Sampling through the reparameterization trick
    def sample(self, mu, logsigma):
        std = torch.exp(logsigma /2)
        eps = torch.rand_like(std)
        return mu + eps* std

    # Decoding: produce image
    def decode(self, z):
        x = F.relu(self.deFC1(z))
        x = x.view(-1, self.nFeatures,(self.imgWidth-6),(self.imgHeight-6))
```

```
        x = F.relu(self.deConv1(x))
        x = torch.sigmoid(self.deConv2(x))
        return x

    # Forward
    def forward(self,x_in):
        mu, logsigma= self.encode(x_in)
        z           = self.sample(mu, logsigma)
        x_out       = self.decode(z)
        return x_out, mu, logsigma
```

[14]:
```
batch_size    = 1024
learning_rate = 1e-3
num_epochs    = 200
```

[15]:
```
train_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(torch.
 ↪Tensor(x_train),torch.Tensor(y_train)),batch_size=batch_size)
test_loader  = torch.utils.data.TensorDataset(torch.Tensor(x_test),torch.
 ↪Tensor(y_test))
```

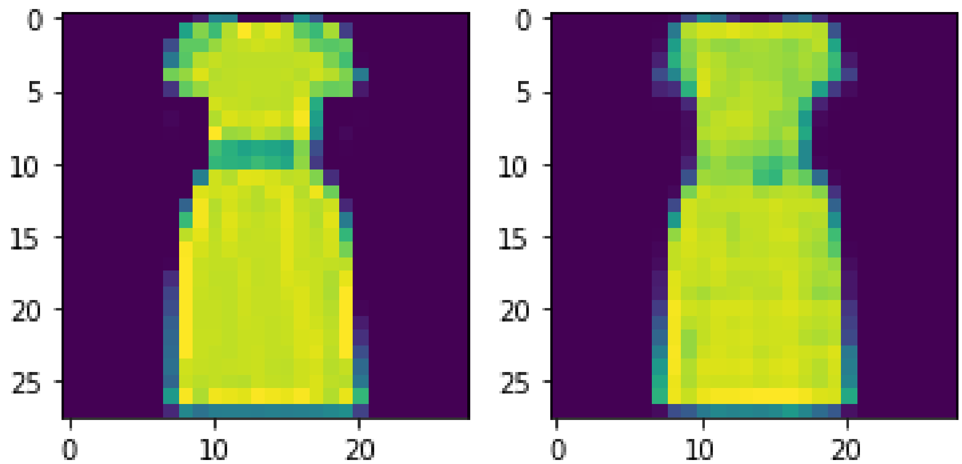
[16]:
```
net = VAE().to(device)
```

[17]:
```
optimizer = torch.optim.Adam(net.parameters(), lr=learning_rate)
```

[ ]:
```
for epoch in range(num_epochs):
    for idx, data in enumerate(tqdm.tqdm(train_loader)):

        imgs, _ = data
        imgs    = imgs.view(-1, 1,  28, 28)
        imgs    = imgs.float()
        imgs    = imgs.to(device)
        out, mu, logVar = net(imgs)

        # loss function
        kl_divergence = 0.5 * torch.sum(-1 - logVar + mu.pow(2) + logVar.exp())
        loss = F.binary_cross_entropy(out, imgs, size_average=False) +␣
 ↪kl_divergence

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    print('Epoch {}: Loss {}'.format(epoch, loss))
```

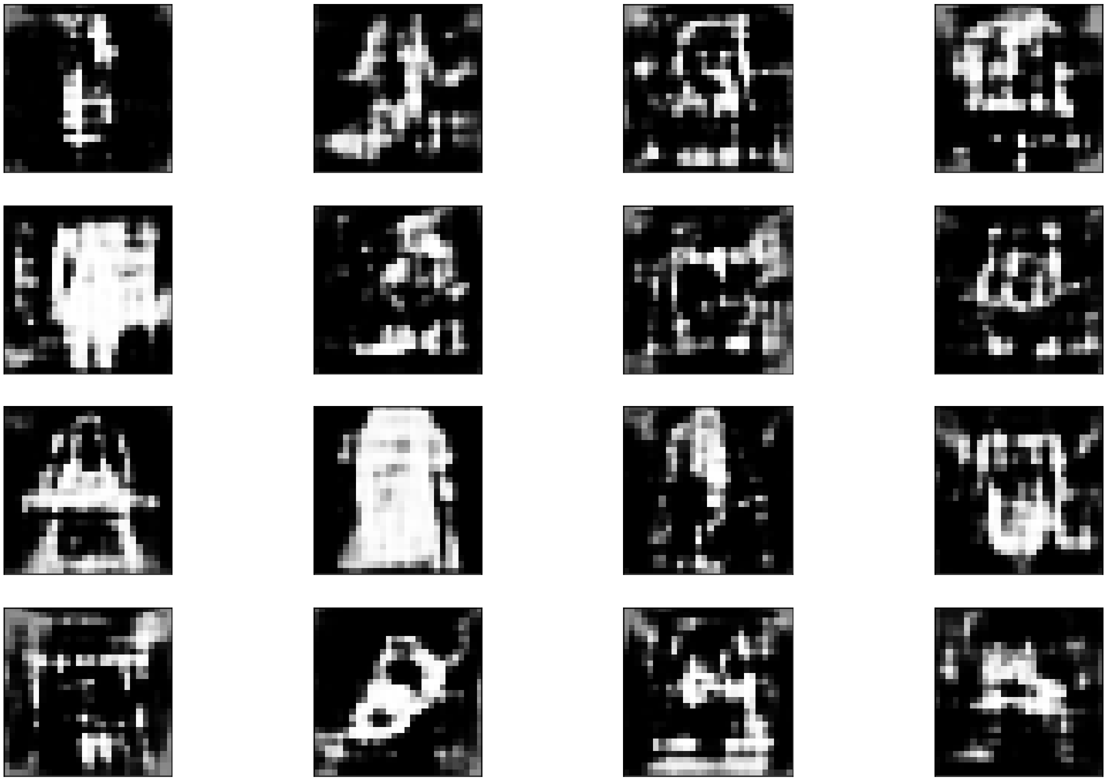
[19]:
```
net.eval()
with torch.no_grad():
    for data in random.sample(list(test_loader), 1):
        imgs, _ = data
```

```
        imgs = imgs.view(-1, 1,  28, 28)
        imgs = imgs.float()
        imgs = imgs.to(device)
        out, mu, logVAR = net(imgs)
        plt.subplot(121)
        plt.imshow(imgs[0, 0].cpu().numpy())
        plt.subplot(122)
        plt.imshow(out[0, 0].cpu().numpy())
        break
```



[21]:

```
net.eval()
plt.figure(figsize=(15, 10))

with torch.no_grad():
    for i in range(16):
        mu, sigma = 0.0,1.0
        z   = torch.Tensor(np.random.normal(mu, sigma, 64))
        z   = z.to(device)
        img = net.decode(z).cpu().numpy()
        plt.subplot(4, 4, i + 1)
        plt.xticks([])
        plt.yticks([])
        plt.imshow(img[0,0],cmap='gray')
```



[72]:
```
net.eval()
dim = 2
with torch.no_grad():
    for data in random.sample(list(test_loader), 1):
        img_in, _ = data
        img_in  = img_in.view(-1, 1,  28, 28)
        plt.figure(figsize=(10, 8))
        plt.subplot(1, 2, 1)
        plt.imshow(img_in[0,0].cpu().numpy(),cmap='gray')
        plt.subplot(1, 2, 2)
        img_in  = img_in.float()
        img_in  = img_in.to(device)
        mu, logsigma= net.encode(img_in)
        img_out  = net.decode(mu)
        plt.imshow(img_out.view(-1, 1,  28, 28)[0,0].cpu().numpy(),cmap='gray')
        plt.show()


        plt.figure(figsize=(15, 10))
        for i,var in enumerate(np.linspace(-1.0,1.0, num=9)):
            plt.subplot(3, 3, i +1 )
            z        = mu
```

```
        z[0,dim]+= var
        img_out  = net.decode(z)
        plt.imshow(img_out[0,0].cpu().numpy(),cmap='gray')
    break
```