Variational Inference

The goal of this notebook is to implement a Variational Autoencoder (VAE).

The combination of a generative model from low dimensional latent variables z to high dimensional observations x (cf. Figure 2), combined with a recognition model from x to z (cf. Figure 1) is what gives rise to the autoencoding structure. The approach we follow here was first proposed by Kingma et. al. [2], although a similar approach was independently proposed by Rezende et. al. [3].

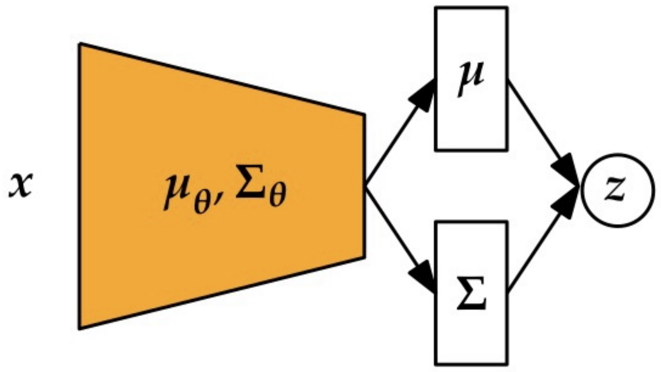


Figure 1. Encoder (inference network $q(z|x)$), provided by M. Deisenroth [1]

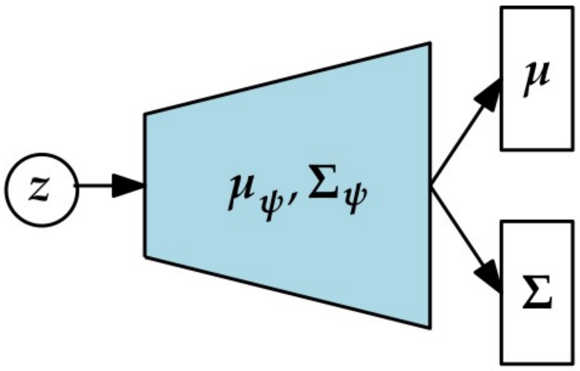


Figure 2. Decoder (generator network $p(x|z)$), provided by M. Deisenroth [1]

As a brief refresher, variational inference is a general approach to doing approximate inference for intractable posterior distributions. For instance, we might have a probabilistic model with observations x and latent variables z and we are interested in computing the posterior distribution $p(z|x)$. For relatively complex data, we might imagine the generative process from $z \to x$ to be some very complicated function we want to learn using a neural network $x = f_\theta(z)$. In order to compute the posterior distribution we need the normalizing factor $p(x) = \int p(x|f_\theta(z))p(z)dz$ which is intractable due to the non-linear mapping we have imposed with the neural network.

Imports

In [1]:
```python
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm_notebook

import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import DataLoader

from torchvision import datasets, transforms
from torchvision.utils import make_grid
```

Setting model and train (hyper)parameters

In [2]:
```python
batch_size = 100
epochs = 20
input_dim = 784
learning_rate = 5e-3
log_interval = 50
num_workers = 8
z_dim = 2

# cuda
cuda = False
device = torch.device("cuda" if cuda and torch.cuda.is_available() else "cpu")
```

Load and visualize MNIST

The MNIST database is a database of handwritten digits that is commonly used for training and testing in the field of machine learning and computer vision. The MNIST database contains 60,000 training images and 10,000 testing images. Each image is of size 28×28.

In [3]:
```python
def load_mnist_dl(batch_size, data_dir='/tmp/data'):
    transf = transforms.ToTensor()
    mnist_train = datasets.MNIST(data_dir, train=True, download=True, transform=transf)
    mnist_test = datasets.MNIST(data_dir, train=False, transform=transf)
    train_loader = DataLoader(mnist_train, batch_size=batch_size, shuffle='True',
                              num_workers=num_workers, drop_last=True)
    test_loader = DataLoader(mnist_test, batch_size=batch_size, shuffle='True',
                             num_workers=num_workers, drop_last=True)
    return train_loader, test_loader
```

In [4]:
```
# get a data loader for each MNIST train and test set
train_loader, test_loader = load_mnist_dl(batch_size)
```

In [5]:
```
# get some random training images
# get the next batch from the train data loader
images, labels = next(iter(train_loader))

# show images from dataset
plt.figure(figsize=(5, 5))
img_grid = make_grid(images, nrow=10)
_ = plt.imshow(img_grid.numpy().transpose(1, 2, 0))
_ = plt.axis('off')    # suppresses axis and its labels
```

Generative Process

Let us consider a dataset $X = \{x_i\}_{i=1}^N$ i.i.d. consisting of N samples of some continuous or discrete variable x. We assume that the data are generated by some random process, involving an unobserved continuous randmo variable z. The process consists of two steps:

1. A value $\boldsymbol{z}_i$ is generated from some prior distribution $p_{\theta^*}(\boldsymbol{z})$.
2. A value $\boldsymbol{x}_i$ is generated from some conditional distribution $p_{\theta^*}(\boldsymbol{x}|\boldsymbol{z})$.

We assumed that the prior $p_{\theta^*}(\boldsymbol{z})$ and likelihood $p_{\theta^*}(\boldsymbol{x}|\boldsymbol{z})$ come from parametric families of distributions $p_\theta(\boldsymbol{z})$ and $p_\theta(\boldsymbol{x}|\boldsymbol{z})$, and that their PDFs are differentiable almost everywehree w.r.t. both θ and $\boldsymbol{z}$. Unfortunatey, a lot of this process is hidden from our view, i.e. both the true parameters θ^* and the values of the latent variables $\boldsymbol{z}_i$ are unknown to us [2].

Let the prior over the latent variables be the centered isotropic multivariate Gaussian $p_\theta(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}\,; \mathbf{0}, \mathbf{I})$. Note taht in this case, the prior lacks parameters. Since in this case we have a binarised MNIST, the observation likelihood is chosen to be Bernoulli $p_\theta(\boldsymbol{x}|\boldsymbol{z}) = \text{Bernoulli}(x; \sigma(\boldsymbol{x}_{\text{logits}}))$ whose distribution parameters are computed from $\boldsymbol{z}$ with a deep fully-connected neural network f_θ to capture the complex generative process of high dimensional observations such as images.

To summarise, for a binarised MNIST we have:

$$\begin{aligned} \boldsymbol{z} &\sim \mathcal{N}(\boldsymbol{z}\,; \mathbf{0}, \mathbf{I}) \\ \boldsymbol{x}_{\text{logits}} &= f_\theta(z) \\ \boldsymbol{x} &\sim \text{Bernoulli}(x; \sigma(\boldsymbol{x}_{\text{logits}})), \end{aligned}$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function.

Just to be clear, logits in mathematics is defined as a function L that maps probabilities $[0, 1] \rightarrow (-\infty, \infty)$, where $p \mapsto L(p) := \log\left(\frac{p}{1-p}\right)$ [4].
The logits have the following properties [4]:

- $p < 0.5 \quad \implies \quad x_{\text{logit}} < 0$
- $p = 0.5 \quad \implies \quad x_{\text{logit}} = 0$
- $p > 0.5 \quad \implies \quad x_{\text{logit}} > 0$

Note, we used $x_{\text{logit}} := L(p)$.

Mean Field Approximation

In this type of variational inference, we assume the variational distribution over the latent variables factorises as

$$q(z_1, \ldots, z_m) = \prod_{j=1}^{m} q(z_j)$$

We refer to $q(z_j)$, the variational approximation for a single latent variable, as a "local variational approxi- mation"[5].

Inference

The objective is to approximate the true posterior distribution $p_\theta(\boldsymbol{z}|\boldsymbol{x}_i)$ which in this case is intractable. While there is much freedom in the form of $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$, we will assume that the true but intractable posterior takes on an approximate Gaussian form with an approximately diagonal covariance. In order to approximate $p_\theta(\boldsymbol{z}|\boldsymbol{x}_i)$, we use the aforementioned mean field variational approximation. In the setting of the variational autoencoder, the variational posterior $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$ is also parameterised by a neural network f_ϕ which takes input $\boldsymbol{x}_i$ and outputs the mean $\boldsymbol{\mu}_i$ and variance $\boldsymbol{\sigma}_i^2$ of the approximate posterior Normal distribution:

$$\begin{aligned}\boldsymbol{\mu}(\boldsymbol{x}_i;\phi),\ \log\boldsymbol{\sigma}^2(\boldsymbol{x}_i;\phi) &= f_\phi(\boldsymbol{x}_i)\\ q_\phi(\boldsymbol{z}|\boldsymbol{x}_i) &= \mathcal{N}(\boldsymbol{z}\,;\ \boldsymbol{\mu}(\boldsymbol{x}_i;\phi),\ \boldsymbol{\sigma}^2(\boldsymbol{x}_i;\phi)\mathbf{I}) \qquad (i)\end{aligned}$$

ELBO (Evidence Lower Bound)

The true posterior has the following form:

$$\begin{aligned}p_\theta(\boldsymbol{z}|\boldsymbol{x}_i) &= \frac{p_\theta(\boldsymbol{z},\boldsymbol{x}_i)}{p_\theta(x_i)}\\ &= \frac{p_\theta(\boldsymbol{z}|\boldsymbol{x}_i)p_\theta(\boldsymbol{z})}{p_\theta(x_i)}\end{aligned}$$

where

$$\begin{aligned}p_\theta(\boldsymbol{z}) &= \mathcal{N}(\boldsymbol{z}\,;\mathbf{0},\mathbf{I})\\ p_\theta(\boldsymbol{x}|\boldsymbol{z}) &= \text{Bernoulli}(x;\sigma(\boldsymbol{x}_{\text{logits}})).\end{aligned}$$

This leaves us only with the marginal likelihood (evidence) $p_\theta(\boldsymbol{x}_i)$ to optimise.

Because directly optimising $\log(p_\theta(\boldsymbol{x}_i))$ is infeasible, we choose to optimise a lower bound $\mathcal{L}$ of it. The lower bound on the marginal likelihood of datapoint $\boldsymbol{x}_i$ can be written as:

$$\begin{aligned}\log(p_\theta(\boldsymbol{x}_i)) \geq \mathcal{L}(x_i;\theta,\phi) &:= E_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\left[\log p_\theta(\boldsymbol{x}_i,\boldsymbol{z}) - \log q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\right]\\ &= E_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\left[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z})\right] - D_{KL}\left(q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\ \|\ p(\boldsymbol{z})\right)\end{aligned}$$

Here you will find the detailed derivation:

$$\begin{aligned}\log(p_\theta(\boldsymbol{x}_i)) =&\log\left(\int p_\theta(x_i|\boldsymbol{z})p_\theta(\boldsymbol{z})dz\right)\\ \overset{\text{importance sampling [6]}}{=}&\log\left(\int p_\theta(x_i|\boldsymbol{z})\frac{p_\theta(\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)dz\right)\\ =&\log\left(E_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\left[\int p_\theta(x_i|\boldsymbol{z})\frac{p_\theta(\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\right]\right)\\ \overset{\text{Jensen's inequality}}{\geq}& E_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\left[\log\left(\int p_\theta(x_i|\boldsymbol{z})\frac{p_\theta(\boldsymbol{z})}{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\right)\right]\\ =&E_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\left[\log\left(p_\theta(\boldsymbol{x}_i|\boldsymbol{z})\right)\right] - E_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\left[\log\left(\frac{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}{p_\theta(\boldsymbol{z})}\right)\right]\\ =&E_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}\left[\log\left(p_\theta(\boldsymbol{x}_i|\boldsymbol{z})\right)\right] - D_{KL}(q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\ \|\ p_\theta(\boldsymbol{z}))\end{aligned}$$

The KL term which acts as a ***regulariser*** can be integrated analytically in our case.
For a general covariance matrix $\Sigma = (\sigma_{ij})_{i,j=1}^N$ and a fixed $i,j \in \{1,\ldots,N\}$ we obtain the following analytical solution for the KL term:

$$\begin{aligned}D_{KL}(q_\phi(z_j|x_i)\ \|\ p_\theta(z_j)) &= E_{q_\phi(z_j|x_i)}\left[\log\left(\frac{q_\phi(z_j|x_i)}{p_\theta(z_j)}\right)\right]\\ &= E_{q_\phi(z_j|x_i)}\left[\log\left(q_\phi(z_j|x_i)\right)\right] - E_{q_\phi(z_j|x_i)}\left[\log\left(p_\theta(z_j)\right)\right]\\ &= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\left(\sigma_{ij}^2\right) - \frac{1}{2} - \left(-\frac{1}{2}\log(2\pi) - \frac{1}{2}\left(\sigma_{ij}^2+\mu_{ij}^2\right)\right)\\ &= \frac{1}{2}\left(\sigma_{ij}^2+\mu_{ij}^2\right) - \frac{1}{2}\log\left(\sigma_{ij}^2\right) - \frac{1}{2}\end{aligned}$$

Therefore the ELBO simplifies for a fixed $i,j \in \{1,\ldots,N\}$ to:

$$\begin{aligned}\log(p_\theta(x_i)) &\geq E_{q_\phi(z_j|x_i)}\left[\log\left(p_\theta(x_i|z_j)\right)\right] - D_{KL}(q_\phi(z_j|x_i)\ \|\ p_\theta(z_j))\\ &= E_{q_\phi(z_j|x_i)}\left[\log\left(p_\theta(x_i|z_j)\right)\right] - \left(\frac{1}{2}\left(\sigma_{ij}^2+\mu_{ij}^2\right) - \frac{1}{2}\log\left(\sigma_{ij}^2\right) - \frac{1}{2}\right)\\ &= E_{q_\phi(z_j|x_i)}\left[\log\left(p_\theta(x_i|z_j)\right)\right] - \frac{1}{2}\left(\sigma_{ij}^2+\mu_{ij}^2\right) + \frac{1}{2}\log\left(\sigma_{ij}^2\right) + \frac{1}{2} \qquad (ii)\end{aligned}$$

Here you will find the step by step analytical KL term solution for a general covariance matrix $\Sigma = (\sigma_{ij})_{i,j=1}^N$ and a fixed $i,j \in \{1,\ldots,N\}$:

$$
\begin{aligned}
D_{KL}(q_\phi(z_j|x_i) \parallel p_\theta(z_j)) &= E_{q_\phi(z_j|x_i)}\left[\log\left(\frac{q_\phi(z_j|x_i)}{p_\theta(z_j)}\right)\right] \\
&= E_{q_\phi(z_j|x_i)}\left[\log\big(q_\phi(z_j|x_i)\big)\right] - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= E_{q_\phi(z_j|x_i)}\left[\log\left(\frac{1}{\sqrt{2\pi\sigma_{ij}^2}}\exp\left(-\frac{1}{2\sigma_{ij}^2}(z_j-\mu_{ij})^2\right)\right)\right] - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= E_{q_\phi(z_j|x_i)}\left[-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2\sigma_{ij}^2}(z_j-\mu_{ij})^2\right] - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= E_{q_\phi(z_j|x_i)}\left[-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2\sigma_{ij}^2}\big(z_j^2 - 2z_j\mu_{ij} + \mu_{ij}^2\big)\right] - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2\sigma_{ij}^2}E_{q_\phi(z_j|x_i)}\left[z_j^2 - 2z_j\mu_{ij} + \mu_{ij}^2\right] - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2\sigma_{ij}^2}\big(E_{q_\phi(z_j|x_i)}\left[z_j^2\right] - 2\mu_{ij}E_{q_\phi(z_j|x_i)}\left[z_j\right] + \mu_{ij}^2\big) - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&\overset{(*)}{=} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2\sigma_{ij}^2}\Big(\mathrm{Var}_{q_\phi(z_j|x_i)}(z_j) + \big(E_{q_\phi(z_j|x_i)}[z_j]\big)^2 - 2\mu_{ij}E_{q_\phi(z_j|x_i)}\left[z_j\right] + \mu_{ij}^2\Big) - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&\overset{(i)}{=} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2\sigma_{ij}^2}\big(\sigma_{ij}^2 + \mu_{ij}^2 - 2\mu_{ij}\cdot\mu_{ij} + \mu_{ij}^2\big) - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2\sigma_{ij}^2}\cdot\sigma_{ij}^2 - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2} - E_{q_\phi(z_j|x_i)}\left[\log\big(p_\theta(z_j)\big)\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2} - E_{q_\phi(z_j|x_i)}\left[\log\left(\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}z_{ij}^2\right)\right)\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2} - E_{q_\phi(z_j|x_i)}\left[-\frac{1}{2}\log(2\pi) - \frac{1}{2}z_{ij}^2\right] \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2} - \left(-\frac{1}{2}\log(2\pi) - \frac{1}{2}E_{q_\phi(z_j|x_i)}\left[z_{ij}^2\right]\right) \\
&= -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2} + \frac{1}{2}\log(2\pi) + \frac{1}{2}E_{q_\phi(z_j|x_i)}\left[z_{ij}^2\right] \\
&\overset{(*)}{=} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2} + \frac{1}{2}\log(2\pi) + \frac{1}{2}\Big(\mathrm{Var}_{q_\phi(z_j|x_i)}(z_j) + \big(E_{q_\phi(z_j|x_i)}[z_j]\big)^2\Big) \\
&\overset{(i)}{=} -\frac{1}{2}\log(2\pi) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2} + \frac{1}{2}\log(2\pi) + \frac{1}{2}\big(\sigma_{ij}^2 + \mu_{ij}^2\big) \\
&= \frac{1}{2}\big(\sigma_{ij}^2 + \mu_{ij}^2\big) - \frac{1}{2}\log\big(\sigma_{ij}^2\big) - \frac{1}{2}
\end{aligned}
$$

$(*)$ We used the following identity:

$$
\begin{aligned}
&\mathrm{Var}_{q_\phi(z_j|x_i)}(z_j) = E_{q_\phi(z_j|x_i)}\left[z_j^2\right] - \big(E_{q_\phi(z_j|x_i)}[z_j]\big)^2 \\
\iff\quad &E_{q_\phi(z_j|x_i)}\left[z^2\right] = \mathrm{Var}_{q_\phi(z_j|x_i)}(z_j) + \big(E_{q_\phi(z_j|x_i)}[z_j]\big)^2
\end{aligned}
$$

The expected ***reconstruction error*** $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$ requires estimation using Monte Carlo by sampling from $q_\phi(z_i|x_i)$:

$$
E_{q_\phi(z_i|x_i)}\left[\log(p_\theta(x_i|z_i))\right] \simeq \frac{1}{L}\sum_{\ell=1}^{L}\log\big(p_\theta\left(x_i|z_i^\ell\right)\big), \qquad z_i^\ell \sim q_\phi(z_i|x_i)
$$

According to the original paper [2], L can be set to 1 if the minibatch size M is large enough ($M \geq 100$).

The ELBO over one batch can be calculated with:

$$
\begin{aligned}
\mathcal{L}(X;\theta,\phi) &= \frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{L}\sum_{\ell=1}^{L}\log\big(p_\theta\left(x_i|z_i^\ell\right)\big) - D_{KL}(q_\phi(z_i|x_i) \parallel p(z_i))\right) \\
&\overset{(ii)}{=} \frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{L}\sum_{\ell=1}^{L}\log\big(p_\theta\left(x_i|z_i^\ell\right)\big) - \frac{1}{2}\big(\sigma_{ii}^2 + \mu_{ii}^2\big) + \frac{1}{2}\log\big(\sigma_{ii}^2\big) + \frac{1}{2}\right)
\end{aligned}
$$

Automatic differentiation

Pytorch uses automatic differentiation (autograd) to automate the computation of backward passes in neural networks. When using autograd, the *forward* pass of your network will define a computational graph; nodes in the graph will be Tensors, and edges will be functions that produce output Tensors from input Tensors. Backpropagating through this graph then allows you to easily compute gradients.

The reparametrisation trick

One of the key aspects of the Kingma et. al. [2] paper was the introduction of the reparametrisation trick. The problem we run into with training the VAE is that we need the gradient of an expectation of the lower bound w.r.t. the parameters of the variational distribution. I.e. $\nabla_\phi\mathcal{L}(x_i;\theta,\phi)$ in the equations above. Without going into detail (see [7] for a great tutorial), we can avoid this problem for some distributions by reparameterising our random variable in terms of a deterministic function and a random variable that is independent of the parameters we wish to take the gradient with respect to.

For a Gaussian distribution, we can get an unbias estimate of the Monte Carlo approximation of the expectations by taking a sample from a normal Gaussian $\epsilon \sim \mathcal{N}(0, I)$ and the reparameterised sample is then given by

$$z_i^l = \mu(x;\phi) + \sigma(x;\phi) * \epsilon$$

You will find the *reparametrise* function in the VAE class below.

In [6]:
```python
class VAE(nn.Module):
    def __init__(self, input_dim=784, z_dim=2):
        super(VAE, self).__init__()
        # Use the nn.Module package to define the VAE model as a sequence of layers.
        # nn.Sequential s a Module which contains other Modules,
        # and applies them in sequence to produce its output.
        # Each Linear module (nn.Linear) computes output from input using a
        # linear function, and holds internal Tensors for its weight and bias.

        # inference network q(z|x)
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU()
        )
        # mean and log(var)
        self.mu = nn.Linear(128, z_dim)
        self.logvar = nn.Linear(128, z_dim)

        # generator network p(x|z)
        self.decoder = nn.Sequential(
            nn.Linear(z_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, input_dim),
            nn.Sigmoid()
        )

    def reparametrise(self, mu, logvar):
        """
        Use mu and log(var) to sample z ~ N(mu, var)
        Parameters
        ----------
        mu:     pytorch Tensor of shape (N, z_dim)
        logvar: pytorch Tensor of shape (N, z_dim)
        Returns
        -------
        z:      pytorch Tensor of shape (N, z_dim)
        """
        # torch.randn_like: Returns a tensor with the same size as input that is filled
        # with random numbers from a normal distribution with mean 0 and variance 1
        e = torch.randn_like(mu)
        std = logvar.exp().sqrt()
        z = mu + e * std
        return z

    def forward(self, x):
        """
        Parameters
        -----------
        x:       pytorch Tensor of shape (N, input_dim)
        Returns
        -------
        recon_x: pytorch Tensor of shape like x
        mu:      pytorch Tensor of shape (N, z_dim)
        logvar:  pytorch Tensor of shape (N, z_dim)
        """
        h = self.encoder(x)
        mu = self.mu(h)
        logvar = self.logvar(h)
        z = self.reparametrise(mu, logvar)
        reconstructed_x = self.decoder(z)
        return reconstructed_x, mu, logvar
```

KL divergence

The KL function returns the KL term between a parametrized Gaussian distribution $q_\phi(z|x) = \mathcal{N}(\mu(x,\phi), \sigma^2(x,\phi))$ and $p_\theta(x) = \mathcal{N}(0,1)$.

In [7]:
```python
def KL(mu, logvar):
    """
    Computes the KL divergence between N(mu, exp(logvar)I) and N(0, I).
    Parameters
    ----------
    mu:     pytorch Tensor of shape (N, D)
    logvar: pytorch Tensor of shape (N, D)
    Returns:
    --------
    kl:     pytorch Tensor of shape (N,)
    """
#     kl = -0.5 * (1 + logvar - mu.pow(2) - logvar.exp())
    kl = 0.5 * (logvar.exp() + mu.pow(2) - logvar - 1)
    return kl.sum(1)
```

Objective function

By changing our objective from maximising the ELBO to the equivalent objective of minimising the negative ELBO, we can use gradient descent methods to optimise variational parameters. The *loss_func* implements the *negative* ELBO for $L = 1$.

$$\mathcal{L}(X;\theta,\phi) = \frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{L}\sum_{\ell=1}^{L} -\log\left(p_\theta\left(x_i|z_i^\ell\right)\right) + D_{KL}(q_\phi(z_i|x_i) \parallel p(z_i))\right)$$

In [8]:
```
def loss_func(recon_x, x, mu, logvar):
    """
    Parameters
    ----------
    mu:      pytorch Tensor of shape (N, z_dim)
    logvar:  pytorch Tensor of shape (N, z_dim)
    recon_x: pytorch Tensor of shape (N, input_dim)
             reconstruction of x
    x: pytorch Tensor of shape (N, input_dim)
    Returns:
    L:       negative ELBO
    -------
    """
    E_log_pxz = F.binary_cross_entropy(recon_x, x, reduction='none').sum(1)
    kl = KL(mu, logvar)
    L = (E_log_pxz + kl).mean()
    return L
```

Init model

In [9]:
```
model = VAE(input_dim=input_dim, z_dim=z_dim).to(device)
```

In [10]:
```
print(model)
```

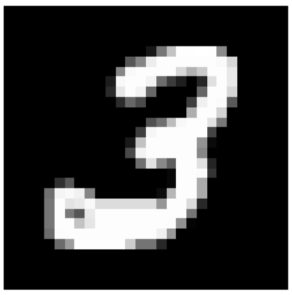
```
VAE(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=128, bias=True)
    (3): ReLU()
  )
  (mu): Linear(in_features=128, out_features=2, bias=True)
  (logvar): Linear(in_features=128, out_features=2, bias=True)
  (decoder): Sequential(
    (0): Linear(in_features=2, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=784, bias=True)
    (5): Sigmoid()
  )
)
```

Little demostration on how torch.bernoulli works

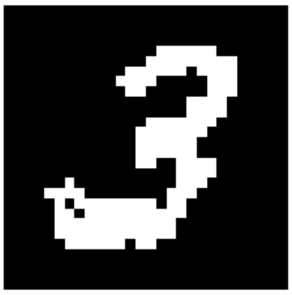
`torch.bernoulli(data)` will return with probability p 1 and zero otherwise.

In [11]:
```
x, _ = next(iter(train_loader))
```

In [12]:
```
plt.figure()
plt.imshow(x[0][0], cmap='gray', interpolation='none')
_ = plt.axis('off')    # suppresses axis and its labels
```



```
In [13]: # Note, if you re-run this cell a couple of times you see that the binarisation is dynamic, i.e.
         # the image still displays the same label as above but varies it a bit in each run
         plt.figure()
         new_x = torch.bernoulli(x)
         plt.imshow(new_x[0][0], cmap='gray', interpolation='none')
         _ = plt.axis('off')    # suppresses axis and its labels
```



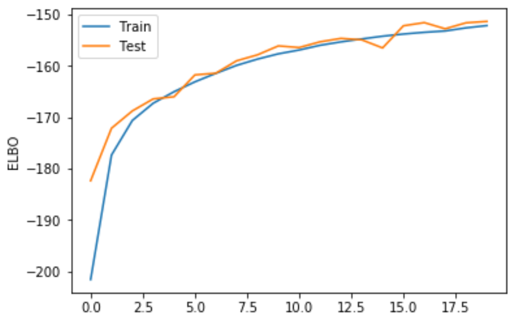
Training and evaluation

```
In [14]: # this should not take more than 5 min to train for 20 epochs

         no_train_batches = len(train_loader)
         no_train_samples = len(train_loader.dataset)
         no_test_samples = len(test_loader.dataset)

         loss_train_history = []
         loss_test_history = []
         ebar = tqdm_notebook(range(epochs), desc='[Epoch {}]'.format(1))
         for epoch in ebar:
             nbar = tqdm_notebook(enumerate(train_loader),
                                  total=no_train_batches,
                                  desc='Training...',
                                  leave=False)
             loss_train = 0.
             model.train()
             for i, (data, labels) in nbar:
                 # dynamically binarise data, cf. demo above
                 data = torch.bernoulli(data).to(device)
                 data = data.view(data.shape[0], -1)

                 # calculate loss using ELBO
                 recon_x, mu, logvar = model(data)
                 loss = loss_func(recon_x, data, mu, logvar)

                 # Zero the gradients before running the backward pass.
                 model.zero_grad()

                 # Backward pass: compute gradient of the loss with respect to all the learnable
                 # parameters of the model. Internally, the parameters of each Module are stored
                 # in Tensors with requires_grad=True, so this call will compute gradients for
                 # all learnable parameters in the model.
                 loss.backward()

                 # Update the weights using gradient descent.
                 # Each parameter is a Tensor, so
                 # we can access its gradients like we did before.
                 with torch.no_grad():
                     for param in model.parameters():
                         param -= learning_rate * param.grad

                 # detach(): Returns a new Tensor, detached from the current graph. This is useful
                 #           if the result will never require its gradient.
                 loss_train += loss.detach().item() * batch_size / no_train_samples
             loss_train_history.append(loss_train)

             # evaluate on test dataset
             model.eval()
             loss_test = 0.
             with torch.no_grad():
                 for i, (data, labels) in enumerate(test_loader):
                     data = torch.bernoulli(data).to(device)
                     data = data.view(data.shape[0], -1)
                     recon_x, mu, logvar = model(data)
                     loss_test += loss_func(recon_x, data, mu, logvar).item() * batch_size / no_test_samples
             loss_test_history.append(loss_test)
             ebar.set_description('[Epoch {}/{}] train: {:.4f} test: {:.4f}'.format(
                 epoch + 1, epochs, loss_train, loss_test))
```

```
[Epoch 20/20] train: 152.1370 test: 151.3254          100% 20/20 [02:26<00:00, 7.40s/it]
```

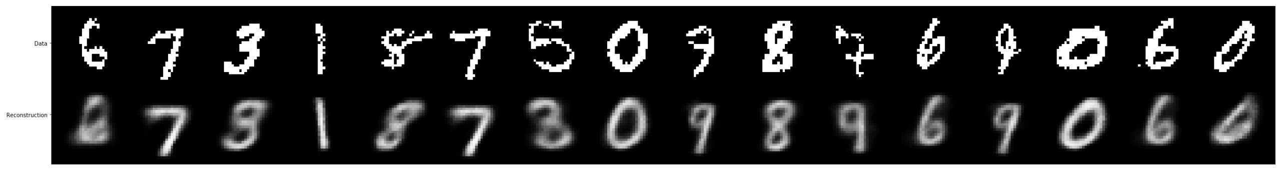
Plot ELBO History

```
In [15]: # ELBO history
         plt.figure()
         _ = plt.plot(range(epochs), np.asarray(loss_train_history) * -1, label='Train')
         _ = plt.plot(range(epochs), np.asarray(loss_test_history) * -1, label='Test')
         _ = plt.ylabel('ELBO')
         _ = plt.legend()
```



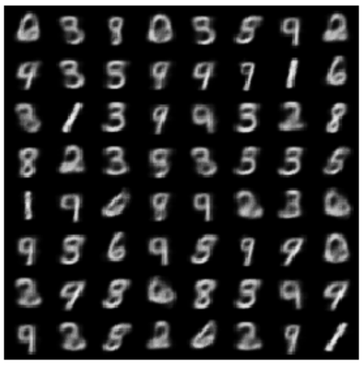
Plot Input vs. Reconstruction

```
In [16]: # plot reconstruction
         n = min(recon_x.shape[0], 16)
         recon_data = torch.cat([data[:n].view(n, 1, 28, 28),
                                 recon_x[:n].view(n, 1, 28, 28)], 0)
         grid_data = make_grid(recon_data, nrow=n)
         plt.figure(figsize=(40, 40))
         _ = plt.imshow(grid_data.numpy().transpose(1, 2, 0), cmap='gray')
         _ = plt.xticks([])
         _ = plt.yticks([14, 42], ['Data', 'Reconstruction'])
```



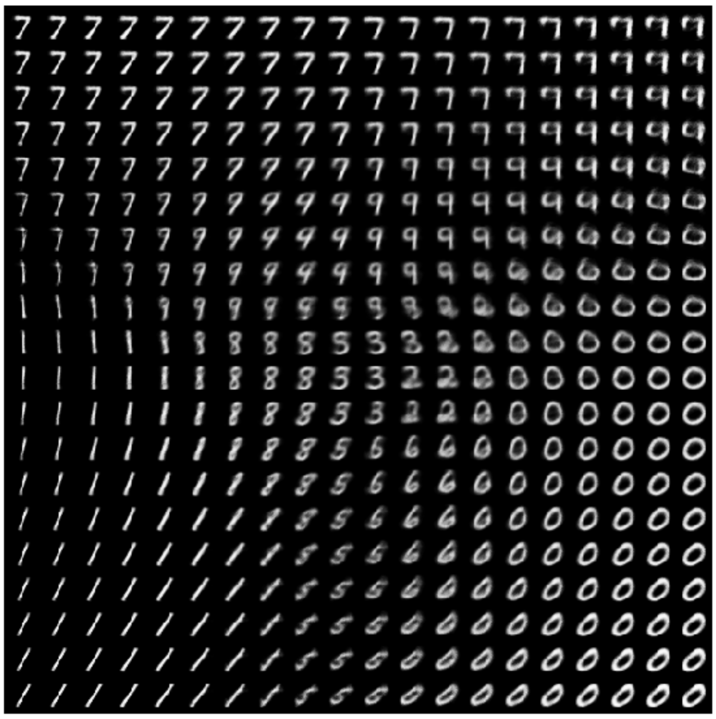
Plot Model Sample

```
In [20]: # sample from model
         z = torch.randn(64, z_dim).to(device)
         with torch.no_grad():
             recon_x = model.decoder(z)

         plt.figure(figsize=(5,5))
         _ = plt.imshow(make_grid(recon_x.view(64, 1, 28, 28)).cpu().numpy().transpose(1, 2, 0))
         _ = plt.axis('off')    # suppresses axis and its labels
```



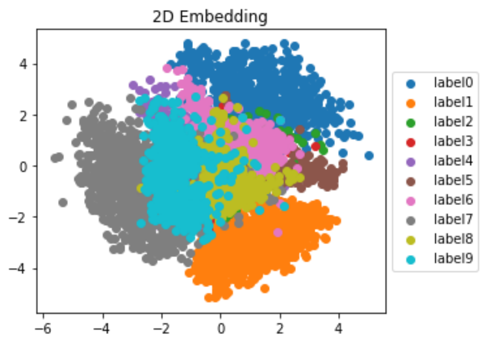
Plot Interpolate Through Latent

In [21]:
```
# interpolate through latent if z_dim = 2
if z_dim == 2:
    plt.figure(figsize=(10, 10))
    x = y = torch.linspace(-4, 4, steps=20)
    xv, yv = torch.meshgrid((x, y))
    z = torch.cat((xv.flatten().unsqueeze(1), yv.flatten().unsqueeze(1)), 1)
    with torch.no_grad():
        recon_x = model.decoder(z)
    _ = plt.imshow(make_grid(recon_x.view(-1, 1, 28, 28), nrow=20).cpu().numpy().transpose(1, 2, 0))
    _ = plt.axis('off')    # suppresses axis and its labels
```



Plot 2D Embedding (Latent Space)

In [23]:
```
### place legend outside
### https://stackoverflow.com/questions/4700614/how-to-put-the-legend-out-of-the-plot
### Accessed: 29/04/2019

z_test = torch.Tensor(no_test_samples, z_dim)
labels_test = []

model.eval()
with torch.no_grad():
    for i, (data, labels) in enumerate(test_loader):
        data = torch.bernoulli(data).to(device)
        data = data.view(data.shape[0], -1)
        recon_x, mu, logvar = model(data)
        z_test[i * batch_size:i * batch_size + data.shape[0], :] = model.reparametrise(mu, logvar)
        labels_test.append(labels)

labels_test = torch.cat(labels_test).numpy()
z_test = z_test.numpy()

# plot latent space if z_dim = 2
if z_dim == 2:
    fig = plt.figure()
    ax = plt.subplot(111)
    plt.title('2D Embedding')

    for y in range(10):
        z_test_y = z_test[labels_test == y]
        ax.scatter(z_test_y[:, 0], z_test_y[:, 1], label='label{}'.format(y))

    # Shrink current axis by 20%
    box = ax.get_position()
    ax.set_position([box.x0, box.y0, box.width * 0.8, box.height])
    # Put a legend to the right of the current axis
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
```



Credits

This notebook is based on the variational inference tutorial from CO493 Probabilistic Inference (spring term 2019) taught by Marc Deisenroth.

References

[1] Imperial College London, CO493 Probabilistic Inference (spring term 2019), Marc Deisenroth, *Variational Inference*
[2] Diederik P. Kingma, Max Welling. *Auto-Encoding Variational Bayes*. 2014. https://arxiv.org/pdf/1312.6114v10.pdf (https://arxiv.org/pdf/1312.6114v10.pdf)
[3] Danilo J. Rezende, Shakir Mohamed, Daan Wierstra. *Stochastic Backpropagation and Approximate Inference in Deep Generative Models*. 2014. https://arxiv.org/pdf/1401.4082.pdf (https://arxiv.org/pdf/1401.4082.pdf)
[4] *What is the meaning of the word logits in TensorFlow?* Accessed: 27/04/2019. https://stackoverflow.com/questions/41455101/what-is-the-meaning-of-the-word-logits-in-tensorflow (https://stackoverflow.com/questions/41455101/what-is-the-meaning-of-the-word-logits-in-tensorflow)
[5] Willie Neiswanger. *Lecture 13 : Variational Inference: Mean Field Approximation*. Accessed: 27/04/2019. https://www.cs.cmu.edu/~epxing/Class/10708-17/notes-17/10708-scribe-lecture13.pdf (https://www.cs.cmu.edu/~epxing/Class/10708-17/notes-17/10708-scribe-lecture13.pdf)
[6] Imperial College London, CO493 Probabilistic Inference (spring term 2019), Marc Deisenroth, *Sampling*
[7] *Machine Learning Trick of the Day (4): Reparameterisation Tricks*. Accessed: 29/04/2019. http://blog.shakirm.com/2015/10/machine-learning-trick-of-the-day-4-reparameterisation-tricks/ (http://blog.shakirm.com/2015/10/machine-learning-trick-of-the-day-4-reparameterisation-tricks/)