EE 046202 - Technion - Unsupervised Learning & Data Analysis

- Formerly 046193

Tal Daniel

Tutorial 10 - Deep Unsupervised Learning - Variational Autoencoder (VAE) - Part 2

Agenda

- Recap
- Implementation in PyTorch
  - Architecture
  - Reparameterization Trick
  - Encoder
  - Decoder
  - Assembling the VAE
  - Loss Function
- Example on MNIST
  - Interpolation in the Latent Space
  - t-SNE on the Latent Space

In [1]:
```
# imports for the tutorial
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time

# pytorch imports
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import torchvision

# sklearn imports
from sklearn.manifold import TSNE
```

Recap

- **Notations**:

  1. X - the data we want to model (e.g. images of dogs)
  2. z - the latent variable (this is the *imagination*, the hidden variable that describes the data, we have seen this before)
  3. $P(X)$ - the probability distribution of the data (e.g. the distribution of all dogs' images in the world)
  4. $P(z)$ - the probability distribution of the latent variables (the source of the imagination, the brain in this case or the distribution of dogs' images feaures/hidden representations). The **prior**.
  5. $P(X|z)$ - the distribution of data generation **given latent variable** (given the features we want the dog to have, the probability of images that satisfy these conditions, turning imagination to real image). The **likelihood**.
- **Objective**:
  - The optimization problem: make the simpler distribution, $Q(z|X)$ as closer as possible to $P(z|X)$.
  - The **KL-divergence** is formulated as follows:

$$D_{KL}[Q(z|X)||P(z|X)] = \mathbb{E}_{Q(z|X)}[\log \frac{Q(z|X)}{P(z|X)}]$$

    - $\mathbb{E}_{Q(z|X)}[\log \frac{Q(z|X)}{P(z|X)}] = \sum_z Q(z|X) \log \frac{Q(z|X)}{P(z|X)}$

$$\mathbb{E}_{Q(z|X)}[\log \frac{Q(z|X)}{P(z|X)}] = \mathbb{E}_{Q(z|X)}[\log Q(z|X) - \log P(z|X)]$$

    Using **Bayes' Rule**:

$$\rightarrow D_{KL}[Q(z|X)||P(z|X)] = \mathbb{E}_{Q(z|X)}[\log Q(z|X) - \log \frac{P(X|z)P(z)}{P(X)}]$$
$$= \mathbb{E}_{Q(z|X)}[\log Q(z|X) - (\log P(X|z) + \log P(z) - \log P(X))]$$
$$= \mathbb{E}_{Q(z|X)}[\log Q(z|X) - \log P(X|z) - \log P(z) + \log P(X))]$$

  - Notice that the expecation is over z and $P(X)$ does not depend on z:

$$\rightarrow D_{KL}[Q(z|X)||P(z|X)] = \mathbb{E}_{Q(z|X)}[\log Q(z|X) - \log P(X|z) - \log P(z)] + \log P(X)$$
$$\log P(X) \geq \mathbb{E}_{Q(z|X)}[\log P(X|z)] - D_{KL}[Q(z|X)||P(Z)] = ELBO$$

  - **Loss Function**:

$$\mathcal{L}_{VAE} = -\mathbb{E}_{Q(z|X)}[\log P(X|z)] + D_{KL}[Q(z|X)||P(z)]$$

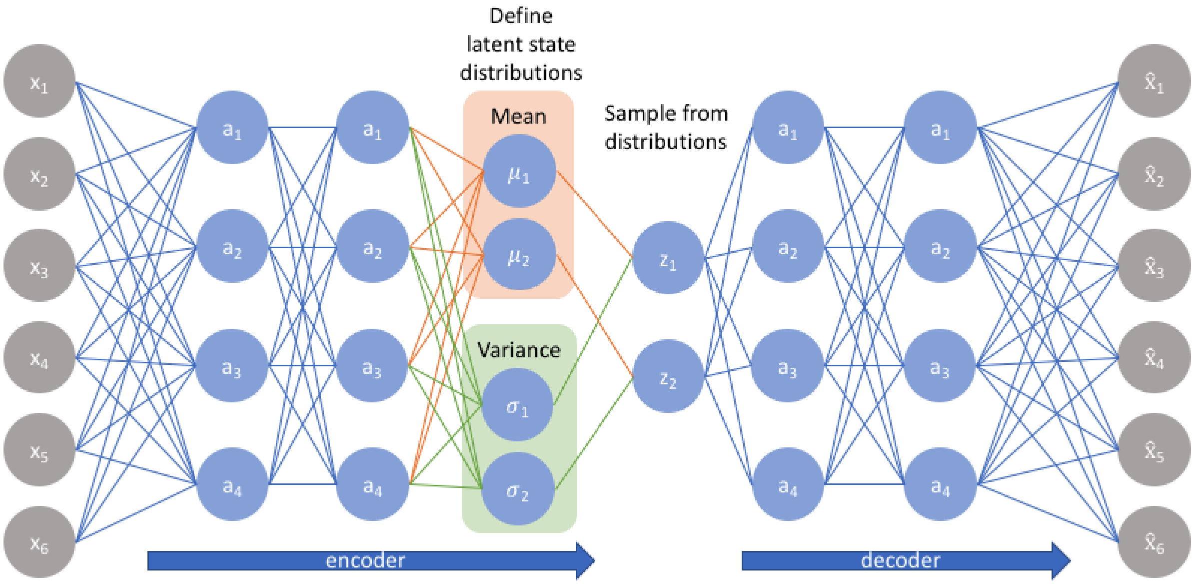
  - **KL-Divergence Closed-form Solution**:
    - Having:
      - $z \sim \mathcal{N}(0, 1)$
      - $z|X \sim \mathcal{N}(\mu(X), \Sigma(X))$

$$D_{KL}[Q(z|X)||P(z)] = \frac{1}{2}\sum_{i=1}^{d}[\Sigma(X)_{ii} + \mu(X)_i^2 - 1 - \log \Sigma(X)_{ii}]$$

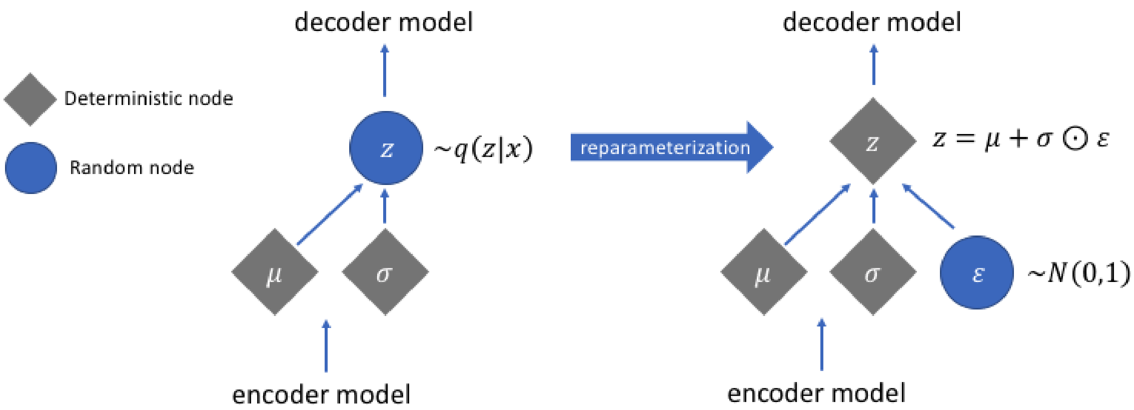
  - **Reconstruction Loss**: $\mathbb{E}_{Q(z|X)}[\log P(X|z)]$ - this is also called the **log-likelihood** of X under z. Maximizing the likelihood is a well-known concept from Machine Learning course, as **Maximum Likelihood Estimation (MLE)**. You have seen this many times in *supervised* learning settings like *Linear Regression* or *Logistic Regression*.

VAE Implementation

We will now implement the VAE components using PyTorch. The general structure of the network:



And remember that we need to implement the **reparameterization trick**:



- Note: we are going to use a simple architecture since we are going to work with a simple dataset, but more difficult datasets require more complex architectures that use convolutional layers, batch normalization layers and etc...

The Reparameterization Trick

We will first implement a function that takes the mean $\mu(x)$ and the variance $\Sigma(X)$ and outputs $z \sim \mathcal{N}(\mu(X), \Sigma(X))$ using the reparemeterization trick so that we can backpropagate the gradients.

In [2]:

```python
# reparametrization trick
def reparameterize(mu, logvar, device=torch.device("cpu")):
    """
    This function applies the reparameterization trick:
    z = mu(X) + sigma(X)^0.5 * epsilon, where epsilon ~ N(0,I)
    :param mu: mean of x
    :param logvar: log variance of x
    :param device: device to perform calculations on
    :return z: the sampled latent variable
    """
    std = torch.exp(0.5 * logvar)
    eps = torch.randn_like(std).to(device)
    return mu + eps * std
```

Encoder

- The encoder takes the high-dimensional data, $X \in \mathcal{R}^D$, and encodes in a lower-dimensional latent space vector, $z \in \mathcal{R}^d$, that is, we model $Q(z|X)$.
- Since we are in a *variational* environment, and we model a distrubution Q, the outputs of the encoder are the mean, $\mu(X) \in \mathcal{R}^d$ and the co-variance, $\Sigma(X) \in \mathcal{R}^d$.
  - Remember that since we assume independce between the latent variables, the co-variance matrix is diagonal and we can represent it as a vector in $\mathcal{R}^d$, where each value represents the variance (the ii^{th} element in the co-variance matrix).

In [3]:

```python
# encoder - Q(z|X)
class VaeEncoder(torch.nn.Module):
    """
       This class builds the encoder for the VAE
       :param x_dim: input dimensions
       :param hidden_size: hidden layer size
       :param z_dim: latent dimensions
       :param device: cpu or gpu
       """

    def __init__(self, x_dim=28*28, hidden_size=256, z_dim=10, device=torch.device("cpu")):
        super(VaeEncoder, self).__init__()
        self.x_dim = x_dim
        self.hidden_size = hidden_size
        self.z_dim = z_dim
        self.device = device

        self.features = nn.Sequential(nn.Linear(x_dim, self.hidden_size),
                                      nn.ReLU())

        self.fc1 = nn.Linear(self.hidden_size, self.z_dim, bias=True)  # fully-connected to output mu
        self.fc2 = nn.Linear(self.hidden_size, self.z_dim, bias=True)  # fully-connected to output logvar


    def bottleneck(self, h):
        """
        This function takes features from the encoder and outputs mu, log-var and a latent space vector z
        :param h: features from the encoder
        :return: z, mu, log-variance
        """
        mu, logvar = self.fc1(h), self.fc2(h)
        # use the reparametrization trick as torch.normal(mu, logvar.exp()) is not differentiable
        z = reparameterize(mu, logvar, device=self.device)
        return z, mu, logvar

    def forward(self, x):
        """
        This is the function called when doing the forward pass:
        z, mu, logvar = VaeEncoder(X)
        """
        h = self.features(x)
        z, mu, logvar = self.bottleneck(h)
        return z, mu, logvar
```

Decoder

- The decoder takes a lower-dimensional latent space vector, $z \in \mathcal{R}^d$ and decodes it to a high-dimensional *reconstruction* data, $\tilde{X} \in \mathcal{R}^D$, that is, we model $P(X|z)$.

In [4]:
```
class VaeDecoder(torch.nn.Module):
    """
       This class builds the decoder for the VAE
       :param x_dim: input dimensions
       :param hidden_size: hidden layer size
       :param z_dim: latent dimensions
       """

    def __init__(self, x_dim=28*28, hidden_size=256, z_dim=10):
        super(VaeDecoder, self).__init__()
        self.x_dim = x_dim
        self.hidden_size = hidden_size
        self.z_dim = z_dim

        self.decoder = nn.Sequential(nn.Linear(self.z_dim, self.hidden_size),
                                     nn.ReLU(),
                                     nn.Linear(self.hidden_size, self.x_dim),
                                     nn.Sigmoid())
        # why we use sigmoid? becaue the pixel values of images are in [0,1] and sigmoid(x) does just tha
t!
        # if you don't work with images, you don't have to use that.


    def forward(self, x):
        """
        This is the function called when doing the forward pass:
        x_reconstruction = VaeDecoder(z)
        """
        x = self.decoder(x)
        return x
```

VAE, Assemble! (Putting It All Together)

We now want to have an end-to-end encoder-decoder model that does everything with one line of code, and we also want the ablity to generate new samples (that is, sample a random vector from the unit normal distribution and decode it - without encoding!).

In [5]:

```python
class Vae(torch.nn.Module):
    def __init__(self, x_dim=28*28, z_dim=10, hidden_size=256, device=torch.device("cpu")):
        super(Vae, self).__init__()
        self.device = device
        self.z_dim = z_dim

        self.encoder = VaeEncoder(x_dim, hidden_size, z_dim=z_dim, device=device)
        self.decoder = VaeDecoder(x_dim, hidden_size, z_dim=z_dim)

    def encode(self, x):
        z, mu, logvar = self.encoder(x)
        return z, mu, logvar

    def decode(self, z):
        x = self.decoder(z)
        return x

    def sample(self, num_samples=1):
        """
        This functions generates new data by sampling random variables and decoding them.
        Vae.sample() actually generates new data!
        Sample z ~ N(0,1)
        """
        z = torch.randn(num_samples, self.z_dim).to(self.device)
        return self.decode(z)

    def forward(self, x):
        """
        This is the function called when doing the forward pass:
        return x_recon, mu, logvar, z = Vae(X)
        """
        z, mu, logvar = self.encode(x)
        x_recon = self.decode(z)
        return x_recon, mu, logvar, z
```

The Loss Function

The loss function is composed of the *reconstruction loss* and the *KL-divergence*:

$$\mathcal{L}_{VAE} = -\mathbb{E}_{Q(z|X)}[\log P(X|z)] + D_{KL}[Q(z|X)||P(z)] = ReconLoss(\tilde{x}, x) + \frac{1}{2}\sum_{i=1}^{d}[\Sigma(X)_{ii} + \mu(X)_i^2 - 1 - \log\Sigma(X)_{ii}]$$

- **Reconstruction Loss**:
  - For images, we will use [Binary Cross Entropy (BCE) (https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy)](https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy), as the values of the pixels are between [0,1] (normalized). There are other alternatives, like "perceptual loss".
  - For continuous inputs, we can use L_1 (MAE) or L_2 (MSE).

In [6]:

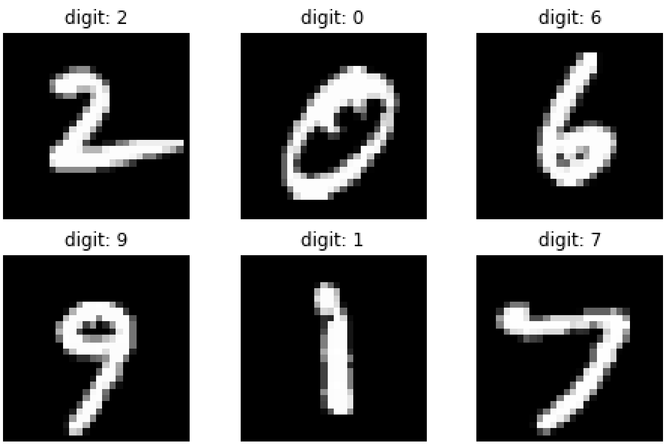
```python
def loss_function(recon_x, x, mu, logvar, loss_type='bce'):
    """
    This function calculates the loss of the VAE.
    loss = reconstruction_loss - 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    :param recon_x: the reconstruction from the decoder
    :param x: the original input
    :param mu: the mean given X, from the encoder
    :param logvar: the log-variance given X, from the encoder
    :param loss_type: type of loss function - 'mse', 'l1', 'bce'
    :return: VAE loss
    """
    if loss_type == 'mse':
        recon_error = F.mse_loss(recon_x, x, reduction='sum')
    elif loss_type == 'l1':
        recon_error = F.l1_loss(recon_x, x, reduction='sum')
    elif loss_type == 'bce':
        recon_error = F.binary_cross_entropy(recon_x, x, reduction='sum')
    else:
        raise NotImplementedError

    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational Bayes. ICLR, 2014
    # https://arxiv.org/abs/1312.6114
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
    kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
    return (recon_error + kl) / x.size(0)
```

Example - VAE On The MNIST Dataset

The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. We will now build the training loop of the VAE and learn an approximation to the hand-written digits distribution.

In [7]:
```
# let's load the dataset and see some examples

# in order to create batches of the data, we create a Dataset and a DataLoader, which takes care of:
# 1. pre-processing the images to tensors with values in [0,1]
# 2. shuffling the data, so we add randomness as learned in ML
# 3. if the data size is not divisible by the batch size, we can drop the last batch
# (so the batches are always of the same size)

# define pre-procesing transformation to use
transform = torchvision.transforms.ToTensor()

train_data = torchvision.datasets.MNIST('./datasets/', train=True, transform=transform,
                                        target_transform=None, download=True)
test_data = torchvision.datasets.MNIST('./datasets/', train=False, transform=transform,
                                       target_transform=None, download=True)



sample_dataloader = DataLoader(train_data, batch_size=6, shuffle=True, drop_last=True)
```

In [8]:
```
fig = plt.figure(figsize=(8 ,5))
samples, labels = next(iter(sample_dataloader))
for i in range(samples.size(0)):
    ax = fig.add_subplot(2, 3, i + 1)
    ax.imshow(samples[i][0, :, :].data.cpu().numpy(), cmap='gray')
    title = "digit: " + str(labels[i].data.cpu().item())
    ax.set_title(title)
    ax.set_axis_off()
```



The VAE Training Loop

We will now build the training loop of the VAE in PyTorch. Pay attention to the order of each function, it is very important in PyTorch.

In [9]:
```
# define hyper-parameters
BATCH_SIZE = 128  # usually 32/64/128/256
LEARNING_RATE = 1e-3  # for the gradient optimizer
NUM_EPOCHS = 150  # how many epochs to run?
HIDDEN_SIZE = 256  # size of the hidden layers in the networks
X_DIM = 28 * 28  # size of the input dimension
Z_DIM = 10  # size of the latent dimension
```

In [10]:

```
# training

# check if there is gpu avilable, if there is, use it
if torch.cuda.is_available():
    torch.cuda.current_device()
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# device = torch.device("cpu")
print("running calculations on: ", device)

# load the data
dataloader = DataLoader(train_data, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)

# create our model and send it to the device (cpu/gpu)

vae = Vae(x_dim=X_DIM, z_dim=Z_DIM, hidden_size=HIDDEN_SIZE, device=device).to(device)

# optimizer
vae_optim = torch.optim.Adam(params=vae.parameters(), lr=LEARNING_RATE)

# save the losses from each epoch, we might want to plot it later
train_losses = []

# here we go
for epoch in range(NUM_EPOCHS):
    epoch_start_time = time.time()
    batch_losses = []
    for batch_i, batch in enumerate(dataloader):
        # forward pass
        x = batch[0].to(device).view(-1, X_DIM)  # just the images
        x_recon, mu, logvar, z = vae(x)
        # calculate the loss
        loss = loss_function(x_recon, x, mu, logvar, loss_type='bce')
        # optimization (same 3 steps everytime)
        vae_optim.zero_grad()
        loss.backward()
        vae_optim.step()
        # save loss
        batch_losses.append(loss.data.cpu().item())
    train_losses.append(np.mean(batch_losses))
    print("epoch: {} training loss: {:.5f} epoch time: {:.3f} sec".format(epoch, train_losses[-1],
                                                                          time.time() - epoch_start_time))
```

```
running calculations on:  cuda:0
epoch: 0 training loss: 171.84777 epoch time: 7.842 sec
epoch: 1 training loss: 128.69239 epoch time: 7.625 sec
epoch: 2 training loss: 122.78445 epoch time: 7.774 sec
epoch: 3 training loss: 119.74014 epoch time: 7.808 sec
epoch: 4 training loss: 117.81168 epoch time: 7.726 sec
epoch: 5 training loss: 116.45555 epoch time: 7.780 sec
epoch: 6 training loss: 115.35806 epoch time: 7.660 sec
epoch: 7 training loss: 114.56222 epoch time: 7.645 sec
epoch: 8 training loss: 113.82657 epoch time: 7.708 sec
epoch: 9 training loss: 113.21288 epoch time: 7.673 sec
epoch: 10 training loss: 112.67187 epoch time: 7.772 sec
epoch: 11 training loss: 112.20706 epoch time: 7.669 sec
epoch: 12 training loss: 111.83337 epoch time: 7.681 sec
epoch: 13 training loss: 111.46434 epoch time: 7.642 sec
epoch: 14 training loss: 111.14702 epoch time: 7.829 sec
epoch: 15 training loss: 110.86055 epoch time: 7.660 sec
epoch: 16 training loss: 110.57215 epoch time: 7.662 sec
epoch: 17 training loss: 110.28147 epoch time: 7.639 sec
epoch: 18 training loss: 110.04906 epoch time: 7.883 sec
epoch: 19 training loss: 109.85175 epoch time: 7.645 sec
epoch: 20 training loss: 109.66326 epoch time: 7.672 sec
epoch: 21 training loss: 109.46986 epoch time: 7.629 sec
epoch: 22 training loss: 109.28426 epoch time: 7.678 sec
epoch: 23 training loss: 109.12090 epoch time: 7.636 sec
epoch: 24 training loss: 108.94051 epoch time: 7.623 sec
epoch: 25 training loss: 108.80147 epoch time: 7.698 sec
epoch: 26 training loss: 108.66099 epoch time: 7.756 sec
epoch: 27 training loss: 108.53297 epoch time: 7.658 sec
epoch: 28 training loss: 108.42440 epoch time: 7.610 sec
epoch: 29 training loss: 108.30579 epoch time: 7.680 sec
epoch: 30 training loss: 108.18119 epoch time: 7.617 sec
epoch: 31 training loss: 108.06430 epoch time: 7.724 sec
epoch: 32 training loss: 107.97036 epoch time: 7.636 sec
epoch: 33 training loss: 107.86317 epoch time: 8.004 sec
epoch: 34 training loss: 107.78109 epoch time: 7.914 sec
epoch: 35 training loss: 107.68400 epoch time: 7.731 sec
epoch: 36 training loss: 107.61546 epoch time: 7.759 sec
epoch: 37 training loss: 107.50388 epoch time: 7.813 sec
epoch: 38 training loss: 107.44395 epoch time: 7.651 sec
epoch: 39 training loss: 107.34560 epoch time: 7.745 sec
epoch: 40 training loss: 107.27498 epoch time: 8.247 sec
epoch: 41 training loss: 107.20559 epoch time: 8.170 sec
epoch: 42 training loss: 107.14346 epoch time: 8.016 sec
epoch: 43 training loss: 107.07263 epoch time: 8.166 sec
epoch: 44 training loss: 107.01250 epoch time: 7.900 sec
epoch: 45 training loss: 106.96018 epoch time: 7.876 sec
epoch: 46 training loss: 106.90081 epoch time: 7.882 sec
epoch: 47 training loss: 106.84647 epoch time: 7.771 sec
epoch: 48 training loss: 106.79369 epoch time: 8.149 sec
epoch: 49 training loss: 106.70217 epoch time: 8.143 sec
epoch: 50 training loss: 106.66743 epoch time: 8.199 sec
epoch: 51 training loss: 106.60985 epoch time: 7.750 sec
epoch: 52 training loss: 106.57071 epoch time: 7.985 sec
epoch: 53 training loss: 106.51574 epoch time: 8.312 sec
epoch: 54 training loss: 106.48122 epoch time: 8.811 sec
epoch: 55 training loss: 106.44102 epoch time: 9.061 sec
epoch: 56 training loss: 106.40275 epoch time: 13.830 sec
epoch: 57 training loss: 106.31404 epoch time: 8.314 sec
epoch: 58 training loss: 106.26953 epoch time: 8.260 sec
epoch: 59 training loss: 106.24339 epoch time: 7.720 sec
epoch: 60 training loss: 106.21802 epoch time: 7.762 sec
epoch: 61 training loss: 106.16079 epoch time: 7.869 sec
epoch: 62 training loss: 106.15153 epoch time: 7.929 sec
epoch: 63 training loss: 106.11558 epoch time: 8.092 sec
epoch: 64 training loss: 106.05807 epoch time: 8.271 sec
epoch: 65 training loss: 106.07709 epoch time: 8.422 sec
epoch: 66 training loss: 106.01839 epoch time: 8.946 sec
epoch: 67 training loss: 105.93994 epoch time: 8.047 sec
epoch: 68 training loss: 105.92013 epoch time: 8.151 sec
epoch: 69 training loss: 105.90470 epoch time: 8.121 sec
epoch: 70 training loss: 105.83137 epoch time: 8.442 sec
epoch: 71 training loss: 105.84895 epoch time: 8.035 sec
epoch: 72 training loss: 105.80781 epoch time: 7.754 sec
epoch: 73 training loss: 105.78372 epoch time: 7.860 sec
epoch: 74 training loss: 105.75193 epoch time: 8.152 sec
epoch: 75 training loss: 105.74861 epoch time: 7.931 sec
epoch: 76 training loss: 105.72887 epoch time: 7.948 sec
epoch: 77 training loss: 105.67473 epoch time: 7.935 sec
epoch: 78 training loss: 105.66093 epoch time: 7.891 sec
epoch: 79 training loss: 105.60130 epoch time: 7.673 sec
```

```
epoch: 80 training loss: 105.63635 epoch time: 7.712 sec
epoch: 81 training loss: 105.53343 epoch time: 7.683 sec
epoch: 82 training loss: 105.53965 epoch time: 8.461 sec
epoch: 83 training loss: 105.54196 epoch time: 7.819 sec
epoch: 84 training loss: 105.48760 epoch time: 7.793 sec
epoch: 85 training loss: 105.46871 epoch time: 7.776 sec
epoch: 86 training loss: 105.44570 epoch time: 7.801 sec
epoch: 87 training loss: 105.42324 epoch time: 7.864 sec
epoch: 88 training loss: 105.40410 epoch time: 8.129 sec
epoch: 89 training loss: 105.41238 epoch time: 7.855 sec
epoch: 90 training loss: 105.36060 epoch time: 7.779 sec
epoch: 91 training loss: 105.34268 epoch time: 7.679 sec
epoch: 92 training loss: 105.33472 epoch time: 7.725 sec
epoch: 93 training loss: 105.31941 epoch time: 7.680 sec
epoch: 94 training loss: 105.25264 epoch time: 8.079 sec
epoch: 95 training loss: 105.25664 epoch time: 7.830 sec
epoch: 96 training loss: 105.23884 epoch time: 7.729 sec
epoch: 97 training loss: 105.24771 epoch time: 7.838 sec
epoch: 98 training loss: 105.18528 epoch time: 7.851 sec
epoch: 99 training loss: 105.17426 epoch time: 7.761 sec
epoch: 100 training loss: 105.16605 epoch time: 7.840 sec
epoch: 101 training loss: 105.17681 epoch time: 7.814 sec
epoch: 102 training loss: 105.13540 epoch time: 7.704 sec
epoch: 103 training loss: 105.11075 epoch time: 7.695 sec
epoch: 104 training loss: 105.10130 epoch time: 7.815 sec
epoch: 105 training loss: 105.04798 epoch time: 7.683 sec
epoch: 106 training loss: 105.04240 epoch time: 7.764 sec
epoch: 107 training loss: 105.08118 epoch time: 7.655 sec
epoch: 108 training loss: 105.01203 epoch time: 7.728 sec
epoch: 109 training loss: 104.99548 epoch time: 7.885 sec
epoch: 110 training loss: 104.96709 epoch time: 7.770 sec
epoch: 111 training loss: 104.96980 epoch time: 7.729 sec
epoch: 112 training loss: 104.97029 epoch time: 7.729 sec
epoch: 113 training loss: 104.94572 epoch time: 7.780 sec
epoch: 114 training loss: 104.93574 epoch time: 7.658 sec
epoch: 115 training loss: 104.91825 epoch time: 7.704 sec
epoch: 116 training loss: 104.90452 epoch time: 7.692 sec
epoch: 117 training loss: 104.89628 epoch time: 7.758 sec
epoch: 118 training loss: 104.84048 epoch time: 7.668 sec
epoch: 119 training loss: 104.82133 epoch time: 7.742 sec
epoch: 120 training loss: 104.82935 epoch time: 7.649 sec
epoch: 121 training loss: 104.81333 epoch time: 7.725 sec
epoch: 122 training loss: 104.78670 epoch time: 7.676 sec
epoch: 123 training loss: 104.76939 epoch time: 7.726 sec
epoch: 124 training loss: 104.78630 epoch time: 7.681 sec
epoch: 125 training loss: 104.75781 epoch time: 7.750 sec
epoch: 126 training loss: 104.75630 epoch time: 7.681 sec
epoch: 127 training loss: 104.71424 epoch time: 7.722 sec
epoch: 128 training loss: 104.73129 epoch time: 7.672 sec
epoch: 129 training loss: 104.69816 epoch time: 7.737 sec
epoch: 130 training loss: 104.70272 epoch time: 7.648 sec
epoch: 131 training loss: 104.70613 epoch time: 7.715 sec
epoch: 132 training loss: 104.68254 epoch time: 7.672 sec
epoch: 133 training loss: 104.63357 epoch time: 7.705 sec
epoch: 134 training loss: 104.65751 epoch time: 7.660 sec
epoch: 135 training loss: 104.62752 epoch time: 7.688 sec
epoch: 136 training loss: 104.60925 epoch time: 7.667 sec
epoch: 137 training loss: 104.59311 epoch time: 7.662 sec
epoch: 138 training loss: 104.59436 epoch time: 7.700 sec
epoch: 139 training loss: 104.60375 epoch time: 7.678 sec
epoch: 140 training loss: 104.54786 epoch time: 7.988 sec
epoch: 141 training loss: 104.56324 epoch time: 8.464 sec
epoch: 142 training loss: 104.56199 epoch time: 7.915 sec
epoch: 143 training loss: 104.51522 epoch time: 7.674 sec
epoch: 144 training loss: 104.53867 epoch time: 7.713 sec
epoch: 145 training loss: 104.51754 epoch time: 7.674 sec
epoch: 146 training loss: 104.47265 epoch time: 7.698 sec
epoch: 147 training loss: 104.48947 epoch time: 7.657 sec
epoch: 148 training loss: 104.52602 epoch time: 7.701 sec
epoch: 149 training loss: 104.47982 epoch time: 8.150 sec
```

In [11]:

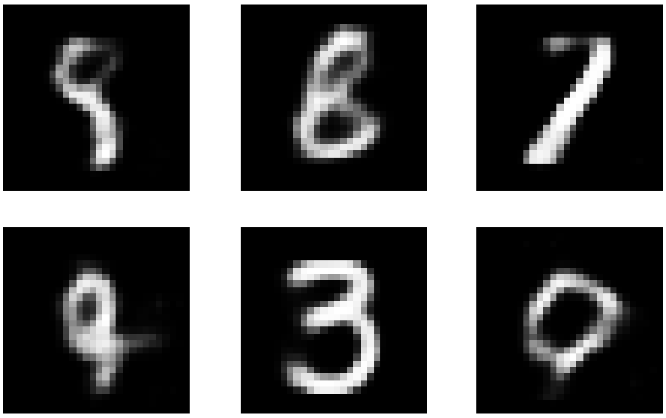
```python
# saving our model (so we don't have to train it again...)
# this is one of the greatest things in pytorch - saving and loading models

# save
fname = "./vae_mnist_" + str(NUM_EPOCHS) + "_epochs.pth"
torch.save(vae.state_dict(), fname)
print("saved checkpoint @", fname)
```

```
saved checkpoint @ ./vae_mnist_150_epochs.pth
```

In [16]:
```python
# Load
vae = Vae(x_dim=X_DIM, z_dim=Z_DIM, hidden_size=HIDDEN_SIZE, device=device).to(device)
vae.load_state_dict(torch.load(fname))
print("loaded checkpoint from", fname)
```

```
loaded checkpoint from ./vae_mnist_150_epochs.pth
```

In [20]:
```python
# now let's sample from the vae
n_samples = 6
vae_samples = vae.sample(num_samples=n_samples).view(n_samples, 28, 28).data.cpu().numpy()
fig = plt.figure(figsize=(8 ,5))
for i in range(vae_samples.shape[0]):
    ax = fig.add_subplot(2, 3, i + 1)
    ax.imshow(vae_samples[i], cmap='gray')
    ax.set_axis_off()
```



Interpolation in the Latent Space

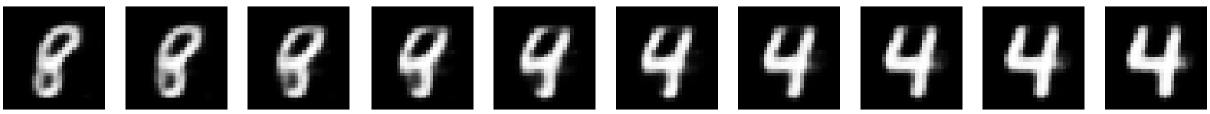
Let's have some fun!

We will take two images, encode them using our VAE and by doing interpolation:

$$z_{new} = \alpha z_1 + (1-\alpha)z_2, \alpha \in [0, 1]$$

we will see the transition between the 2 images.

In [21]:
```python
# let's do something fun - interpolation of the latent space
alphas = np.linspace(0.1, 1, 10)
# take 2 samples
sample_dataloader = DataLoader(test_data, batch_size=2, shuffle=True, drop_last=True)
it = iter(sample_dataloader)
samples, labels = next(it)
while labels[0] == labels[1]:
    # make sure they are different digits
    samples, labels = next(it)
x_1, x_2 = samples

# get their latent representation
_,_, _, z_1 = vae(x_1.view(-1, X_DIM).to(device))
_,_, _, z_2 = vae(x_2.view(-1, X_DIM).to(device))

# let's see the result
fig = plt.figure(figsize=(15 ,8))
for i, alpha in enumerate(alphas):
    z_new = alpha * z_1 + (1 - alpha) * z_2
    x_new = vae.decode(z_new)
    ax = fig.add_subplot(1, 10, i + 1)
    ax.imshow(x_new.view(28, 28).cpu().data.numpy(), cmap='gray')
    ax.set_axis_off()
```

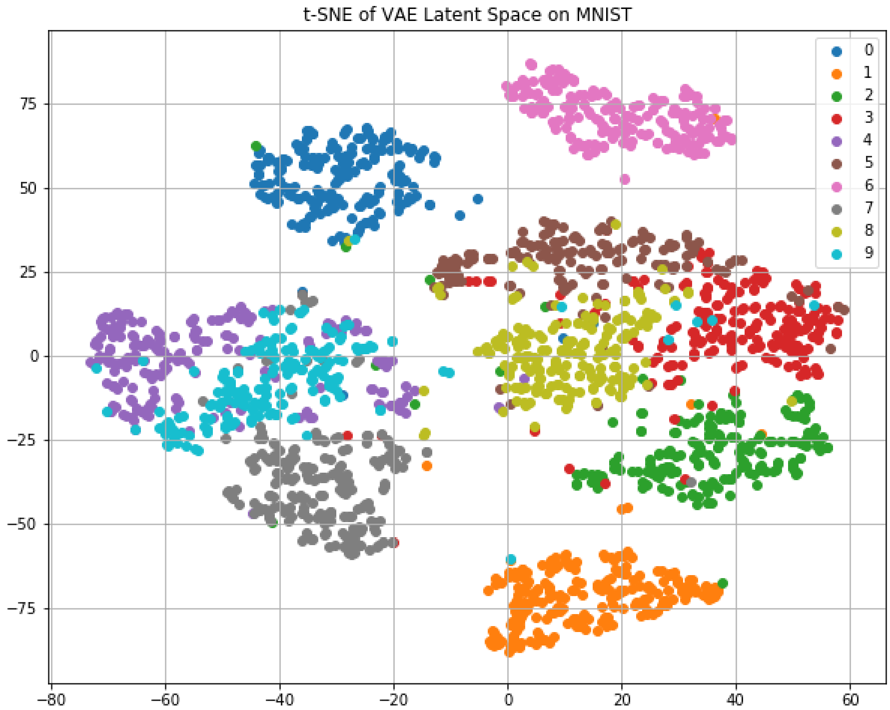


👓 Latent Space Representation with t-SNE

Let's see how descriptive is the latent space. We will take 2000 images, decode them, and reduce their dimensionality with t-SNE.

In [15]:
```
# take 2000 samples
num_samples = 2000
sample_dataloader = DataLoader(train_data, batch_size=num_samples, shuffle=True, drop_last=True)
samples, labels = next(iter(sample_dataloader))

labels = labels.data.cpu().numpy()
# decode the samples
_,_, _, z = vae(samples.view(num_samples, X_DIM).to(device))

# t-SNE
perplexity = 15.0
t_sne = TSNE(n_components=2, perplexity=perplexity)
z_embedded = t_sne.fit_transform(z.data.cpu().numpy())

# plot
fig = plt.figure(figsize=(10 ,8))
ax = fig.add_subplot(1, 1, 1)
for i in np.unique(labels):
    ax.scatter(z_embedded[labels==i,0], z_embedded[labels==i, 1], label=str(i))
ax.legend()
ax.grid()
ax.set_title("t-SNE of VAE Latent Space on MNIST")
```

Out[15]: `Text(0.5, 1.0, 't-SNE of VAE Latent Space on MNIST')`



🏅 Credits

- Deep Learning - Unsupervised Learning, [Tutorial by Ruslan Salakhutdinov (CMU) (https://www.cs.cmu.edu/~rsalakhu/talk_MLSS_part2.pdf)](https://www.cs.cmu.edu/~rsalakhu/talk_MLSS_part2.pdf) - [https://www.cs.cmu.edu/~rsalakhu/ (https://www.cs.cmu.edu/~rsalakhu/)](https://www.cs.cmu.edu/~rsalakhu/)
- [CS294-158 Deep Unsupervised Learning Spring 2019 (https://sites.google.com/view/berkeley-cs294-158-sp19/home)](https://sites.google.com/view/berkeley-cs294-158-sp19/home) @ UC Berkeley - [https://sites.google.com/view/berkeley-cs294-158-sp19/home (https://sites.google.com/view/berkeley-cs294-158-sp19/home)](https://sites.google.com/view/berkeley-cs294-158-sp19/home)
- [Variational autoencoders. (https://jeremyjordan.me/variational-autoencoders/)](https://jeremyjordan.me/variational-autoencoders/) by Jeremy Jordan - [https://jeremyjordan.me (https://jeremyjordan.me)](https://jeremyjordan.me)
- [Variational Autoencoder: Intuition and Implementation (https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/)](https://wiseodd.github.io/techblog/2016/12/10/variational-autoencoder/) by Agustinus Kristiadi
- Icons from [Icon8.com (https://icons8.com/)](https://icons8.com/) - [https://icons8.com (https://icons8.com)](https://icons8.com)
- Datasets from [Kaggle (https://www.kaggle.com/)](https://www.kaggle.com/) - [https://www.kaggle.com/ (https://www.kaggle.com/)](https://www.kaggle.com/)