Date: April 22th, 2019
Team - Akila Jeeson Daniel, Myriam Laiymani, Marc-André Piché, Khalil Slimi

Link to code repository `https://github.com/mennaML/IFT6135H19_assignment3`

Problem 1

Generative Adversarial Network (GAN) discriminators estimate a metric between two distributions. In this question, you will implement a discriminator that estimates the Jensen Shannon divergence (JSD) and a critic that estimates the Wasserstein Distance (WD). Then, using your estimators, you will compare the properties the JSD and the WD.

We provide samplers [1] to generate the different distributions you will need for this question. You can use any architecture as long as your function have enough capacity to correctly estimate the distribution. A 3-layers MLP should suffice. You may use SGD with a learning rate of $1e-3$ and a mini batch size of 512.

1. (10 pts) Implement a function to estimate the Jensen Shannon Divergence. Rembember, the JSD is defined as $D_{JS}(p||q) = \frac{1}{2}D_{KL}(p||r) + \frac{1}{2}D_{KL}(q||r)$ where $r(x) = \frac{p+q}{2}$. Hence, the objective function that your neural network should optimize is:

$$\arg\max_{\theta} \left\{ \log 2 + \frac{1}{2}\mathbf{E}_{x\sim p}[\log(D_\theta(x))] + \frac{1}{2}\mathbf{E}_{y\sim q}\log(1 - D_\theta(y))] \right\}$$

2. (10 pts) Implement a function to estimate the Wasserstein Distance. Rembember, that the dual form of the WD is defined as $W(p,q) = \sup_{||T||_L \leq 1} \mathbf{E}_{x\sim p}[T(x)] - \mathbf{E}_{y\sim q}[T(y)]$. In order to constrain your function to be 1-lipschitz, we recommend that you use Gradient Penalty. Hence, the objective that your network should optimize is

$$\arg\max_{\theta} \mathbf{E}_{x\sim p}[T_\theta(x)] - \mathbf{E}_{y\sim q}[T_\theta(y)] - \lambda \mathbf{E}_{z\sim r}(||\nabla_z T_\theta(z)||_2 - 1)^2.$$

   r is the distribution over $z = ax + (1-a)y$, where $x \sim p$, $y \sim q$ and $a \sim U[0,1]$. We recommend $\lambda \geq 10$.

3. (5 pts) Let $Z \sim U[0,1]$ be a random variable with a uniform distribution. Let p be the distribution of $(0,\ Z)$ and q_θ be the distribution of (ϕ, Z), where ϕ is a parameter. Plot the estimated JSD and the WD for $\phi \in [-1,1]$ with intervale of 0.1 (i.e. 21 points). The x-axis should be the value of ϕ and the y-axis should be your estimate.

[1] See the assignment repository `IFT6135H19_assignment/assignment3/samplers.py`

4. (5 pts) Let f_0 be the density of a 1-dimensional standard Gaussian, and f_1 be the unknown density function of `distribution4`. Train a discriminator D_θ by maximizing the value function

$$\mathbb{E}_{x\sim f_1}[\log D_\theta(x)] + \mathbb{E}_{y\sim f_0}[\log(1 - D_\theta(y))]$$

Estimate the density f_1 using the procedure of Question 5 in the theory part. Plot the discriminator output and the estimated density (using the script `density_estimation.py`).

1. See repository - `https://github.com/mennaML/IFT6135H19_assignment3`
2. See repository - `https://github.com/mennaML/IFT6135H19_assignment3`
3. See Figure.1.

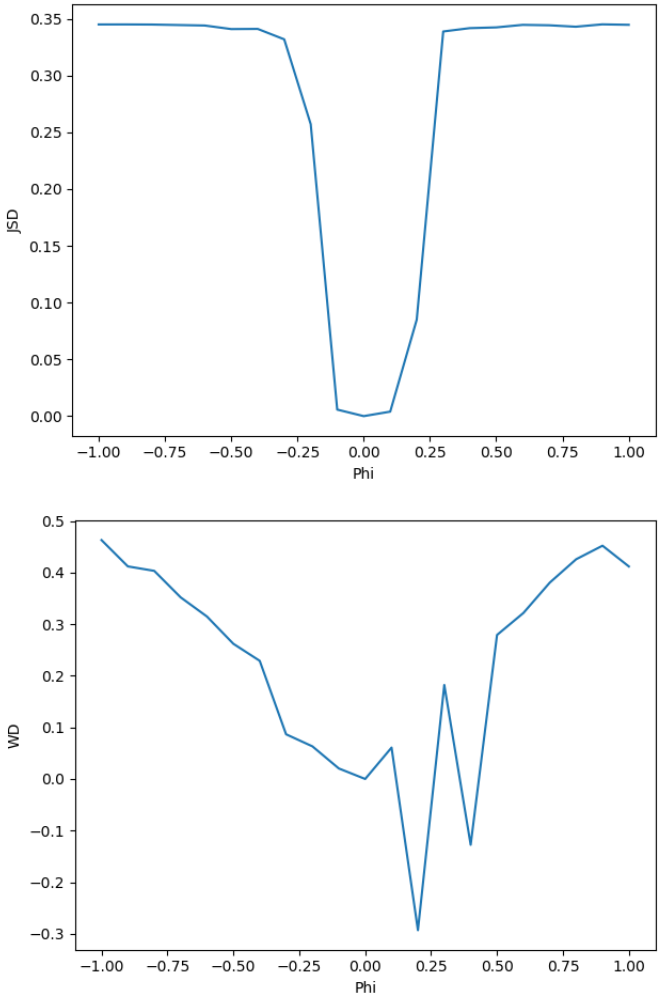

Figure 1: JS distance vs ϕ (top) and Wasserstein distance vs ϕ (bottom)

4. See figure.2

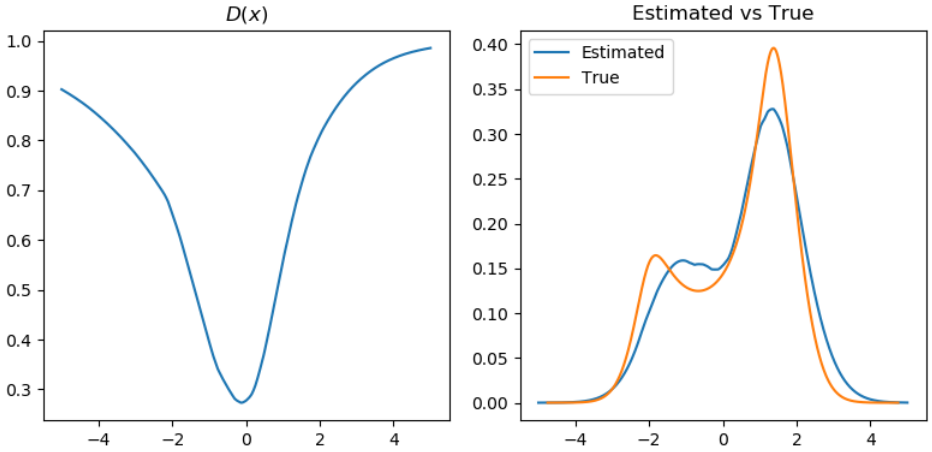

Figure 2: Disriminator output (left) and estimator density (right)

Problem 2

Variational Autoencoders (VAEs) are probabilistic generative models. This means they can be used to estimate $p(\boldsymbol{x})$.

Train a VAE on the *Binarised MNIST* dataset, using the negative ELBO loss as shown in class. The prior $p(\boldsymbol{z}) = \mathcal{N}(\boldsymbol{z}; \mathbf{0}, \mathbf{I})$. Each pixel in this dataset is binary: The pixel is either black or white, which means you have to model the likelihood $p(\boldsymbol{x}|\boldsymbol{z})$, i.e. the decoder, as a product of bernoulli distributions (this means you should use the binary cross entropy loss for reconstruction) over the pixels. This is not interchangeable with the standard MNIST dataset available on `torchvision`, so either (a) Modify the code provided with this assignment or, (b) download the dataset here: `https://github.com/yburda/iwae/tree/master/datasets/BinaryMNIST`, and implement your own data loader.

Train a VAE (10pts) Train a VAE with a latent variable of 100-dimensions. The following are suggested hyperparameters for training the VAE on Binarized MNIST. You may use the following architecture:

Encoder (ϕ)

Conv2d(1, 32, kernel size=(3, 3))
ELU()
AvgPool2d(kernel size=2, stride=2)
Conv2d(32, 64, kernel size=(3, 3))
ELU()
AvgPool2d(kernel size=2, stride=2)
Conv2d(64, 256, kernel size=(5, 5))
ELU()
Linear(in features=256, out features=2 × 100)
(This final layer should output mean and log-variance)

Decoder (θ)

Linear(in features=100, out features=256)
ELU()
Conv2d(256, 64, kernel size=(5, 5), padding=(4, 4))
ELU()
UpsamplingBilinear2d(scale factor=2, mode=bilinear)
Conv2d(64, 32, kernel size=(3, 3), padding=(2, 2))
ELU()
UpsamplingBilinear2d(scale factor=2, mode=bilinear)
Conv2d(32, 16, kernel size=(3, 3), padding=(2, 2))
ELU()
Conv2d(16, 1, kernel size=(3, 3), padding=(2, 2))

This is not meant to be code. Find out the relevant API for the same behaviour in the framework you are familiar with to implement this.

Use ADAM with a learning rate of 3×10^{-4}, and train for 20 epochs. Evaluate the model on the validation set using the **ELBO**. Marks will neither be deducted nor awarded if you do not use the given architecture. Just note that for this question you have to:

1. Achieve an average per-instance ELBO of ≥ -96 on the validation set:
$$\frac{1}{|\mathcal{D}_{\text{valid}}|} \sum_{\boldsymbol{x}_i \in \mathcal{D}_{\text{valid}}} \mathcal{L}_{\text{ELBO}}(\boldsymbol{x}_i) \geq -96$$
2. Report the ELBO of your model.

Feel free to modify the above hyperparameters (except the latent variable size) to ensure it works.

Evaluating log-likelihood with Variational Autoencoders (20 pts) VAE models are *evaluated* with log-likelihood, approximated by importance sampling, which was covered during the lecture. The formula is reproduced here with additional details:

$$\log p(\boldsymbol{x} = \boldsymbol{x}_i) \approx \log \frac{1}{K} \sum_{k=1}^{K} \frac{p_\theta(\boldsymbol{x} = \boldsymbol{x}_i | \boldsymbol{z}_i^{(k)})\ p(\boldsymbol{z} = \boldsymbol{z}_i^{(k)})}{q_\phi(\boldsymbol{z} = \boldsymbol{z}_i^{(k)} | \boldsymbol{x}_i)}; \quad \text{for all } k\text{: } \boldsymbol{z}_i^{(k)} \sim q_\phi(\boldsymbol{z} | \boldsymbol{x}_i)$$

and $\boldsymbol{x}_i \in \mathcal{D}$.

1. With M as the size of the *minibatch* evaluated, $K = 200$ as the number of importance samples being used, D as the dimension of the input ($D = 784$ in the case of MNIST), and $L = 100$ as the dimension of the latent variable, implement this importance sampling procedure as a function that:
   - **Given:**
     - Your model (which was trained in the first part of the question).
     - An (M, D) array of $\boldsymbol{x}_i$'s.
     - An (M, K, L) array of $\boldsymbol{z}_{ik}$'s.
   - **Returns:**
     - $(\log p(\boldsymbol{x}_1), \ldots, \log p(\boldsymbol{x}_M))$ estimates of size $(M,)$

   Display this snippet of code in your report.

   Tip: Watch out for numerical underflow[2] issues (probabilities of images can get really small). Use the LogSumExp[3] trick to deal with this.
2. Report your evaluations of the trained model on the validation and test set using, (a) the ELBO, and (b) the log-likelihood estimate ($\frac{1}{N} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i)$), where N is the size of the dataset.

Make sure your code is linked and available for this question. The points are awarded for correct implementation.

[2] the generation of a number that is too small to be represented in the device meant to store it.
[3] `http://blog.smola.org/post/987977550/log-probabilities-semirings-and-floating-point`

Link to code repository See folder 'Exercise 2' in repository - `https://github.com/mennaML/IFT6135H19_assignment3`

Train a VAE

1. As seen in the figure below, We achieved an average per-instance ELBO of ≥ -96 on the validation set, after training for 20 epochs, using the provided architecture and the following hyper-parameters:

   (a) Batch size: 32

   (b) Batch size: ADAM with a learning rate of 3×10^{-4}

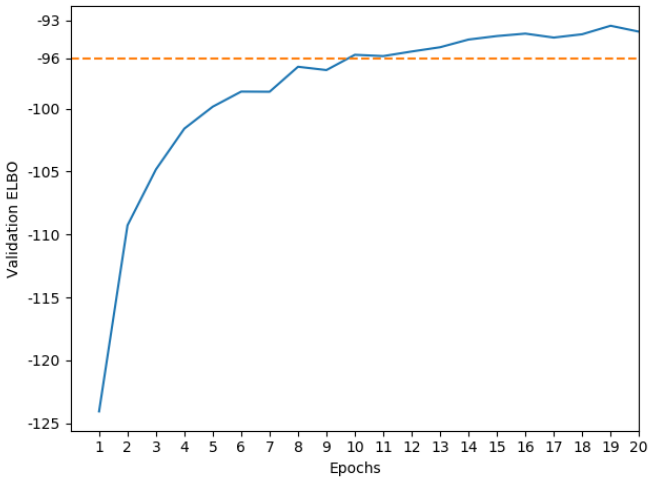

Figure 3: ELBO per epoch on validation set

2. The model obtained an ELBO of **-93.88** on the validation set, after 20 epochs.

Evaluating log-likelihood with Variational Autoencoders

1. Snippet code for the importance sampling procedure:

```
import numpy as np
import torch
import torch.distributions as tdist
from torch.nn import functional as F

# Generates z_ik, log_p_zik and log_q_zik
def generate_samples(model, x_batch, num_samples, device):

    mean_batch, logvar_batch = model.encode(x_batch)
```

```
    mv_normal = tdist.multivariate_normal.MultivariateNormal

    p_z_dist = mv_normal(torch.zeros(mean_batch.size(1)).to(device),
                         torch.eye(mean_batch.size(1)).to(device))

    z_samples = torch.empty((mean_batch.size(0),
                             num_samples, mean_batch.size(1)),
                             device=device)
    log_q_z = torch.empty((mean_batch.size(0), num_samples), device=device)
    log_p_z = torch.empty((mean_batch.size(0), num_samples), device=device)

    for i in range(len(x_batch)):
        q_z_dist = mv_normal(mean_batch[i],
                             torch.diag(torch.exp(0.5*logvar_batch[i])))
        z_i = q_z_dist.sample((num_samples,))
        log_q_z_i = q_z_dist.log_prob(z_i)
        log_p_z_i = p_z_dist.log_prob(z_i)

        z_samples[i] = z_i
        log_q_z[i] = log_q_z_i
        log_p_z[i] = log_p_z_i


    return z_samples, log_p_z, log_q_z

# Calculates the importance sampling approximation of log_p_x over a batch
def estimate_batch_log_density(model, x_batch, num_samples, device):

    z_samples, log_p_z, log_q_z = generate_samples(model, x_batch,
                                                   num_samples, device)

    result = torch.empty((len(x_batch), ), device=device)
    for i in range(len(x_batch)):

        x_predict = model.decode(z_samples[i])
        log_p_z_i = log_p_z[i]
        log_q_z_i = log_q_z[i]

        log_p_x_z_i = torch.empty((num_samples, ), device=device)

        for k in range(num_samples):
            log_p_x_z_ik = -F.binary_cross_entropy(
            x_predict[k].view(-1, 784),
            x_batch[i].view(-1, 784),
            reduction='sum')
```

```
                log_p_x_z_i[k] = log_p_x_z_ik.item()

            logsum = log_p_x_z_i + log_p_z_i - log_q_z_i

            logpx = -np.log(num_samples) + torch.logsumexp(logsum, 0)
            result[i] = logpx

        return result
```

2. On validation set, we obtain :
   (a) ELBO = **-93.88**
   (b) $\log p(x) \approx$ **-88.79**
3. On test set, we obtain :
   (a) ELBO = **-93.10**
   (b) $\log p(x) \approx$ **-88.25**

Problem 3

Recent years have shown an explosion of research into using deep learning and computer vision algorithms to generate images. An ongoing challenge in the image generation community is the question of evaluation. In computer science we often want objective metrics by which to compare our algorithms, but for aesthetic tasks it is not obvious that such a metric even exists. Thus, assessing the quality of a generative model often involves a combination of qualitative and quantitative evaluations, and evaluation of generative models is an active area of research.

In this final question, we investigate the advantages and disadvantages of two popular generative models in deep learning: GANs and VAEs, using some popular evaluation methods from the literature.

Street View House Numbers (SVHN) The SVHN[4] dataset is made up of clipped images of house numbers from Google Street View. Like MNIST, the goal is to classify digits (0-9), but it is a larger dataset, with color images.

The data is available here: `http://ufldl.stanford.edu/housenumbers/train_32x32.mat` (train set), and `http://ufldl.stanford.edu/housenumbers/test_32x32.mat` (test set).

To compare VAE and GAN we used models that had almost the same architecture for encoder and discriminator, and the same architecture for decoder and generator.

[4] `http://ufldl.stanford.edu/housenumbers/`

- Encoder/Discriminator : 4 blocks of [convolution, batch norm and LeakyReLU] that go from 32 feature maps of 16x16 to 256 of 2x2. The only difference being the last step to the output shape being done with a linear layer for the VAE and last round of convolution with sigmoid for the discriminator.
- Decoder/Generator : The decoder/generator is also made of 4 blocs of [transposed convolution, batch norm and ReLU] going from 256 feature maps of 2x2 to 32 of 16x16. The final step to 3x32x32 is made with a transposed convolution followed by tanh.

Both models have around 1,6M parameters. We put a lot of effort into upgrading this DCGAN with the WGAN-GP objective but found only worst results and performance, so we reverted back to our initial model with BCE objective.

Qualitative Evaluation We want to evaluate the "goodness" of the generative models and if the latent space "makes sense".

Visual inspecion of both models can be seen in the **SVHN_Visual Colab Notebook**

1. **Provide visual samples.** Comparing both samples of generated images (Figure 2) we immediately notice differences along those aspects:
   (a) **Sharpness:** VAE's images are blurrier while GAN's image sharper but also have more noise in the colour spectrum
   (b) **Represented digits:** VAE almost always have at least the shape of digits in the image, even if blurry, while GAN suffers from mode failure and often has nothing recognizable in the image.
   (c) **Distortion:** GAN's digits suffer from distortion in shape and colour while VAE's are much straighter and more even.
   (d) **Diversity:** Aside from instances of mode failure, GAN is showing greater diversity in number of digits show, colour and shape

Figure 4: Generated images for GAN (left) and VAE (right).

2. **Disentangled representation in the latent space.** Sampling a random z from our prior distribution, we made small perturbations for each dimension ($z_i' = z_i + \epsilon$). Some interesting results for GAN and VAE are shown in Figure 3 and 4. For GAN, all 100 dimensions showed changes, however, not all $G(z)$ reacted equally to changes, some z are more resistant than others to changes. For the changes to be noticeable, we had to use a large epsilon (0.8) that covers most of the range of z $[z_i min \approx -2.5, z_i max \approx 2.5]$ over 5 steps. In contrast, any sample of VAE reacted to changes in epsilon. However, some dimensions seemed to have much greater impact on the outcome while most seemed to be unused parameters.

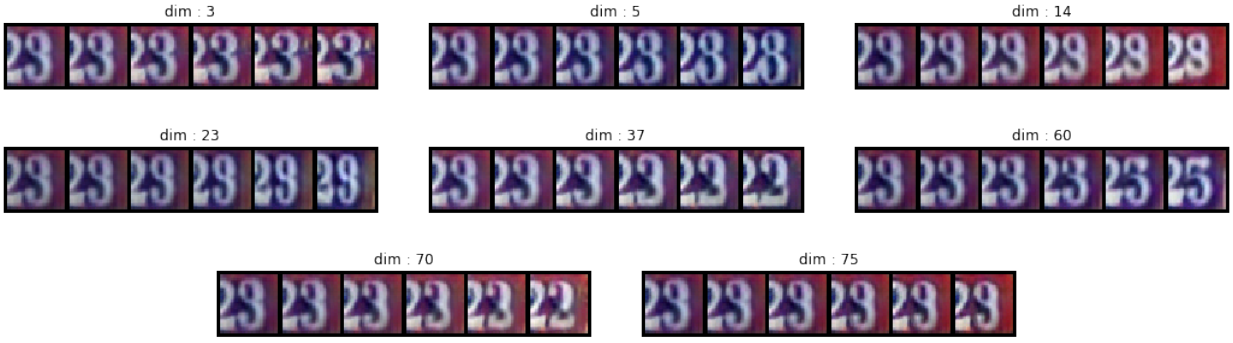

Figure 5: Generated images for GAN for variations in different latent axes.

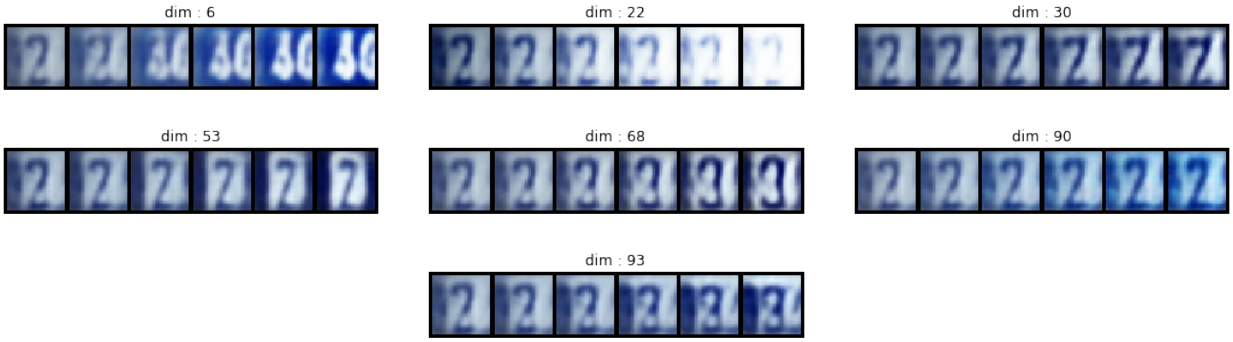

Figure 6: Generated images for VAE for variations in different latent axes.

3. **Interpolating in the data and latent space.** Looking at interpolation in the latent and data spaces, we can see both models varying their representations by morphing from one shape to the other. VAE's morphing is perfectly gradual and even while GAN's go through unnecessary or failure states. Here we are interpolating in the meaningful space for both models. We can clearly see the difference with interpolating is the data space where it is simply a linear transition pixel-to-pixel as both images are superposed with an alpha transparency.

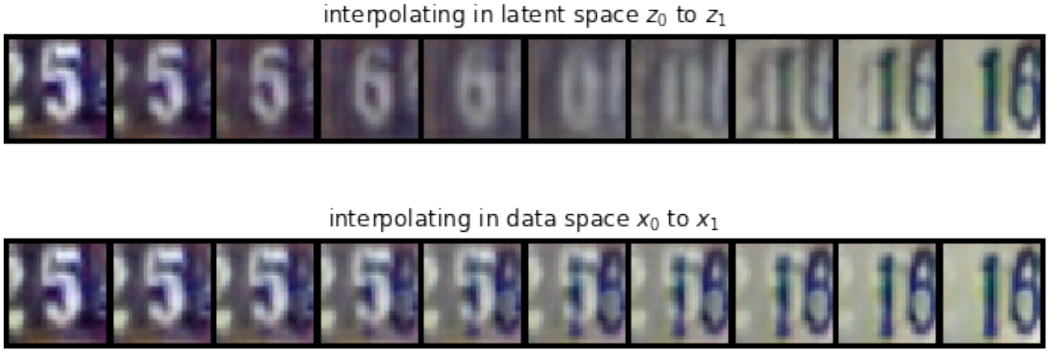

Figure 7: Generated images for GAN for interpolation in latent space and data space.

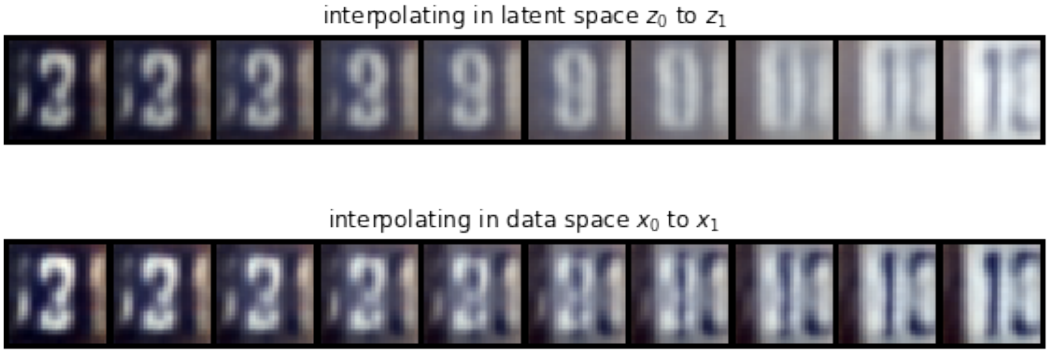

Figure 8: Generated images for VAE for interpolation in latent space and data space.

Quantitative Evaluations Some of the best generative models don't have a suitable tractable quantitative evaluation metric (they just provide samples). There are many metrics used in the research community to evaluate generated samples.

The Frechet Inception Distance (FID) is a score commonly used in the literature to evaluate GAN generation. In essence, it computes the l_2 distance of the first and the second order moment between the generated and the target representation. In practice, the representation is the coding layer of a neural network pre-trained on a classification task.

More formally, let p be the target distribution and q be the generator's/decoder's distribution. (μ_p, Σ_p) and (μ_q, Σ_q) are the mean and covariance of the "representations" of p and q, respectively. The FID is defined as

$$d^2((\mu_p, \Sigma_p), (\mu_q, \Sigma_q)) = ||\mu_p - \mu_q||_2^2 + \mathrm{Tr}(\Sigma_p + \Sigma_q - 2(\Sigma_p \Sigma_q)^{1/2}) \quad (1)$$

Here, we want to evaluate the samples given from both the GAN and the VAE using FID.

1.
```
def calculate_fid_score(sample_feature_iterator,
                        testset_feature_iterator):
    sample_data = []
    for i in sample_feature_iterator:
        sample_data.append(i)
    sample_data = np.array(sample_data)
    mu_sample = np.mean(sample_data, axis=0)
    cov_sample = np.cov(sample_data, rowvar=False)
    sample_data = 0

    testset_data = []
    for i in testset_feature_iterator:
        testset_data.append(i)
    testset_data = np.array(testset_data)
    mu_testset = np.mean(testset_data, axis=0)
    cov_testset = np.cov(testset_data, rowvar=False)
    testset_data = 0

    mu_diff = mu_sample - mu_testset

    eps=1e-4
    offset = np.eye(cov_sample.shape[0]) * eps
    covmean, _ = linalg.sqrtm((cov_sample +
                        offset).dot(cov_testset + offset), disp=False)
    fid_score = mu_diff.dot(mu_diff) + np.trace(cov_sample) +
                        np.trace(cov_testset) - 2*np.trace(covmean)
    return fid_score
```

2. FID score for GAN: 66320.81 and FID score for VAE: 59293.31. Both the FID scores are in the similar range.