Report Practical(Assignment 3)

Team members:
Fares Ben Slimane
Parviz Haggi
Mohammed Loukili
Jorge A. Gutierrez Ortega

April 22, 2019

Abstract

This report includes our solutions to the problems of the 3rd practical assignment. It consists of three sections: In Section 1 we implement the original GAN and WGAN with gradient penalty along with some experimentation. Section 2 is about the implementation and experimentation of the VAEs. The last section (3) includes experimentation and evaluation of the generative models' ability to generate realistic-looking images. In each case, our code is uploaded to the following Github repository [1]

Problem 1

In this problem we are using a GAN discriminator to estimate the Jensen Shannon divergence (JSD) and a the Wasserstein Distance (WD). We implement the discriminator/critic using a multi-layer perceptron that we can initialize to either have an output $y \in [0, 1]$ (sigmoid activation) or an output $y \in \mathbb{R}$ (no activation). The architecture consists of two hidden layers with 64 and 128 output units respectively. We use ReLU activation function in the hidden layers.

1. In this section, we implement a function to estimate the JSD. Using our MLP with sigmoid activation at the output we optimize the following objective function:

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

At its optimum, the objective function estimates the JSD between the distributions given by p and q. An overview of the implementation can be seen below. The full code is available in the file **density_estimation.py** under our github repository [1].

```

#Implementation of the JSD

optimizer.zero_grad()
p = torch.cat((p1,torch.rand(batch_size,1)),1).to(device)
q = torch.cat((phi*torch.ones(batch_size,1),t
                        orch.rand(batch_size,1)),1).to(device)
Dp = net(p)
Dq = net(q)
loss = -(math.log(2.) + (1/2.)*torch.mean(torch.log(Dp)) +
                                                   (1/2.)*torch.mean(torch.log(1-Dq)))
loss.backward()
optimizer.step()
```

2. In this section, we implement a function to estimate the WD. In this case we use our MLP without activation function, so it has an output of a real valued scalar. Here we optimize the following objective function:

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

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

   At it's optimum, this objective function estimates the WD between the distributions given by p and q. The following portion of the code shows the implementation of the optimization. The full code is also available in our repository [1]

```

#Implementation of the WD

optimizer.zero_grad()
p = torch.cat((p1,torch.rand(batch_size,1)),1).to(device)
q = torch.cat((phi*torch.ones(batch_size,1),
                        torch.rand(batch_size,1)),1).to(device)
Dp = net(p)
Dq = net(q)
#  gradient penalty
a = torch.rand(batch_size, 1).expand(batch_size,2).to(device)
r = a*p + (1-a)*q
r.requires_grad = True
Dr = net(r)
gradients = torch.autograd.grad(outputs=Dr, inputs=r,
                grad_outputs=torch.ones(batch_size,1).to(device),
                create_graph=True, retain_graph=True, only_inputs=True)[0]
loss = -(torch.mean(Dp) - torch.mean(Dq) -
         gp_coeff*torch.mean((gradients.norm(2, dim=1) - 1) ** 2))
loss.backward()
optimizer.step()
```

3. In this example we compare the properties of JSD and WD. Given a random variable $Z \sim U[0,1]$, we compute the approximation of both metrics between the 2-dimensional distributions, p given by $(0, Z)$, and q_ϕ given by (ϕ, Z), where ϕ is a parameter. In Figures 1 and 2 we plot the estimated JSD and WD respectively for $\phi \in [-1,1]$ with interval of 0.1. We perform the estimation by optimizing the MLP as in points 1 and 2. We use batches of samples from the distributions of 512, and the models were trained for 5000 iterations using an SGD optimizer. For every value of ϕ we generate the distribution q_ϕ and measure its distance to p.

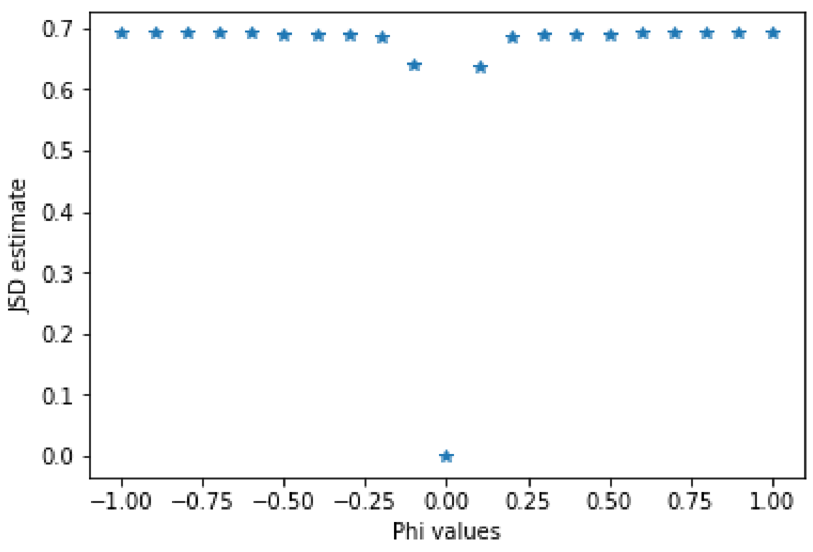

Figure 1: JSD estimation

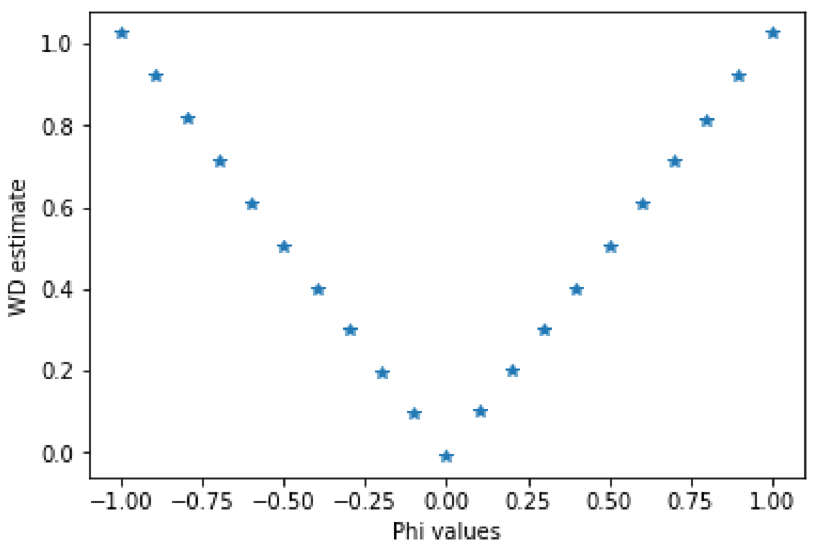

Figure 2: WD estimation

In this experiment the distributions p and q_ϕ have disjoint supports for all values of ϕ but $\phi = 0$. As expected, the JSD when the distributions are disjoint is the same (around $\log(2)$) no matter how close the distributions may be i.e. for a small value of ϕ. This results in a difficulty to learn the distribution p using JSD. In contrast, WD is continuous over the values of ϕ giving a better information about the *closeness* of the distributions.

The full code is given in the file **density_estimation.py** in our github repository [1].

4. In this section we estimate the unknown density f_1 using the approximation $f_0(x)D(x)/(1-D(x))$ (proven in Question 5 from the theoretical part), where f_0 is a known distribution (assumed 1-dimensional standard Gaussian in this question). The full code is provided in the file **density_estimation.py** under our github repository [1].

   Using the above neural network (discriminator), we minimize the following function:

$$loss = -(torch.mean(torch.log(Dx)) + torch.mean(torch.log(1-Dy))),$$

   where Dx is the feedforward of f_1 and Dy is the feedforward of f_0.

   The following figures show the discriminator's output and the estimated density:

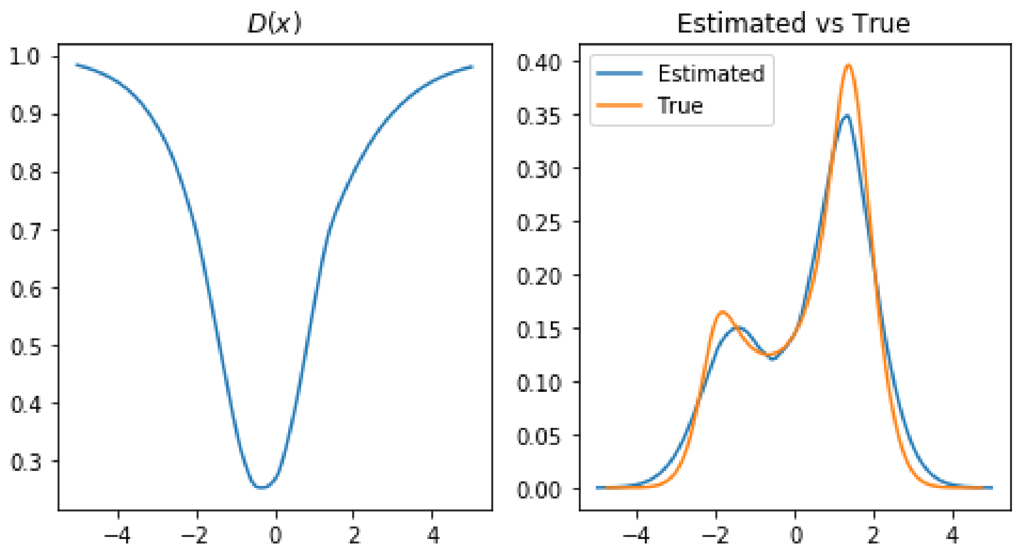

Figure 3: (left) Discriminator output (Right)Estimated f_1

Problem 2

A. Training VAE
We used the given architecture and ADAM with the provided learning rate. After training the model for 20 Epochs, we achieved an average value of ELBO of -95 on validation set. It is clearly higher than the reference value provided in question. The code for training the VAE can be found in script vae.py under the folder Problem2.

B.1 Evaluating log-likelihood with VAE
Here we implement the Importance Sampling procedure that takes as parameters the trained model, an array of x_i and an array of samples z_{ik} from the distribution $q(z|x_i)$. The procedure returns an array of log-likelihood $\log p(x_i)$ of the size of the mini-batches.The code snippet below demonstrates our implementation.

```
##Evaluate VAE using importance sampling
#Q2.1

def loss_IS(model, true_x, z):

  #Loop over the elements i of batch
  M = true_x.shape[0]

  #Save logp(x)
  logp_x = np.zeros([M])

  #Get mean and std from encoder
  #2 Vectors of 100
  mu, logvar = model.encode(true_x.to(device))
  std = torch.exp(0.5*logvar)

  K = 200

  #Loop over tha batch
  for i in range(M):
    #z_ik
    samples = z[i,:,:]

    #Compute the reconstructed x's from sampled z's
    x = model.decode(samples.to(device))

    #Compute the p(x_i|z_ik) of x sampled from z_ik
    #Bernoulli dist = Apply BCE
    #Output an array of losses

    true_xi = true_x[i,:,:].view(-1, 784)
    x = x.view(-1, 784)

    #P(x/z) is shape (200, 784)
    p_xz = -(true_xi * torch.log(x) + (1.0-true_xi) * torch.log(1.0-x))

    #Multiply the independent probabilities (784)
    #Apply logsumexp to avoid small image prob
    #P(x/z) is shape (200, 784)
    p_xz = logsumexp(p_xz.cpu().numpy(), axis=1)


    ##q(z_ik/x_i) follows a normal dist
    #q_z = mgd(samples, mu, std)
    s = std[i, :].view([std.shape[1]])
    m = mu[i, :].view([std.shape[1]])

    q_z = multivariate_normal.pdf(samples.cpu().numpy(),mean=m.cpu().numpy(),
     cov=np.diag(s.cpu().numpy()**2))


    ##p(z_ik) follows a normal dist with mean 0/variance 1
    #Normally distributed with loc=0 and scale=1
    std_1 = torch.ones(samples.shape[1])
    mu_0 = torch.zeros(samples.shape[1])

    p_z = multivariate_normal.pdf(samples.cpu().numpy(),mean=mu_0.cpu().numpy
    (), cov=np.diag(std_1.cpu().numpy()**2))

    #Multiply the probablities

    #logp_x[i] = np.log((1.0/K) * np.sum(np.exp(np.log(p_xz) + np.log(p_z) -
    np.log(q_z))))
    #logp_x[i] = np.log((1.0/K) * np.sum(np.exp(p_xz.cpu().numpy() + np.log(
    p_z) - np.log(q_z))))
    logp_x[i] = np.log((1.0/K) * np.sum((p_xz * p_z)/q_z))
```

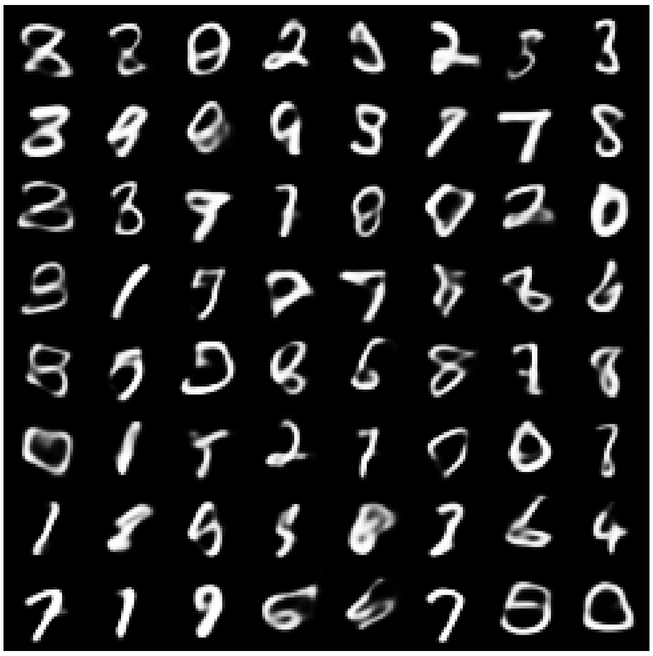
Figure 4: A sample of generated images

```

    return logp_x
```

B.2 The evaluation of the training model using the ELBO:

a. Validation: −95

b. Test: −94.43

The evaluation of the training model using the log-likelihood:

a. Validation: −18.41

b. Test: −18.27

Below (Figure 4) is a sample of the obtained images generated by the trained model:

Problem 3

We have used in this problem a similar architecture for the VAE's decoder and the GAN's generator which is an MLP with 6 layers, as shown in this code snippet:

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.ReLU()
            nn.Linear(128, 256),
            nn.ReLU()
            nn.Linear(256, 512),
            nn.ReLU()
            nn.Linear(512, 1024),
            nn.ReLU()
            nn.Linear(1024, 2048),
            nn.ReLU()
            nn.Linear(2048, int(np.prod(img_shape))),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        img = img.view(img.shape[0], *img_shape)
        return img
```

We have decided to go with this architecture, after having tested several architectures for both models, including convolutionnal neural network. We noticed that the VAE model works fine with a convolutional architecture whereas the GAN have not got good result with that kind of architecture. Both models have been trained with a latent variable dimension of 100.

A. Qualitative Evaluations

1. Visual samples
   The scripts used to generate the samples for the Qualitative evaluation can be found in the QualitativeVAE.py and QualitativeGAN.py under folder Problem 3 for both VAE and GAN models. We have generated different samples from both models (Figures 5 and 6). We notice that the images generated by the VAE seem realistic albeit blurry, whereas the images generated by GAN are more diversified, and seem less realistic.

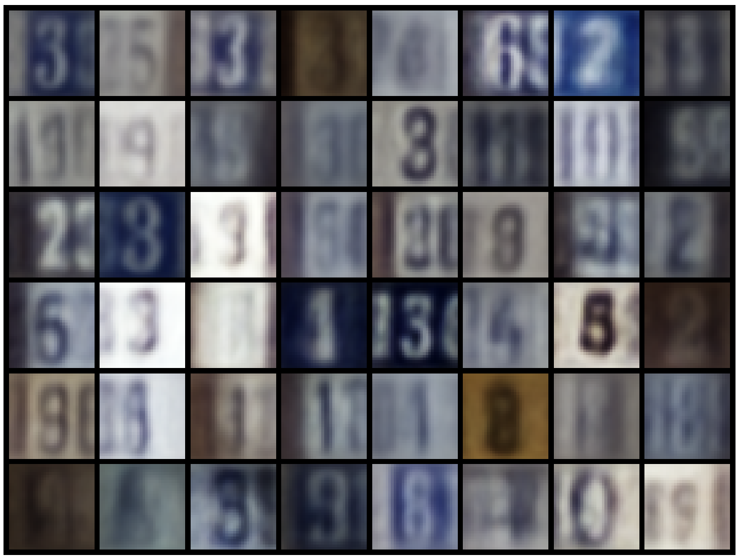

Figure 5: Samples generated with VAE

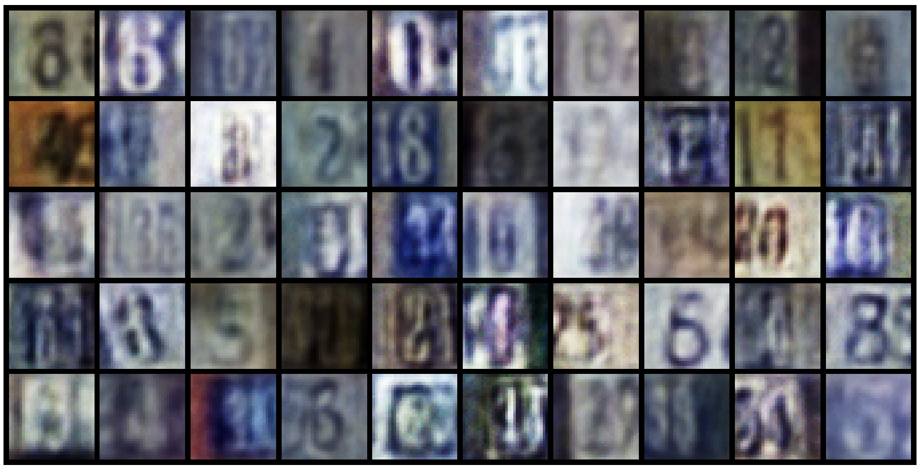
Figure 6: Samples generated with GAN.

2. Learning the disentangled representation in the latent space. Figures 7 and 8 show how the models learned a disentangled representation. Starting from one point in the latent space we apply a perturbation to two arbitrary dimensions. We can observe how in both models the resulting images vary smoothly. The value of perturbation (ϵ) is however different for each model, as they require different levels of perturbation to result in visible changes.

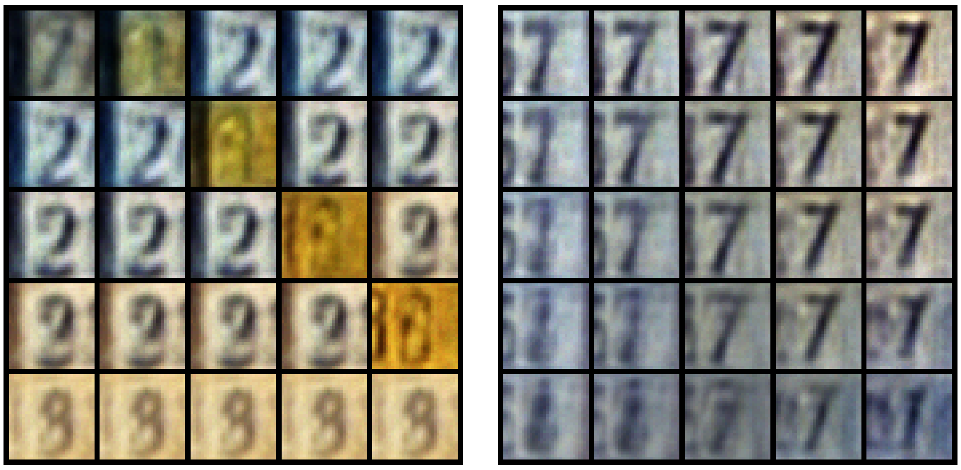
Figure 7: Samples by perturbating the dimensions, left: 6 and 96, right: 25 and 75

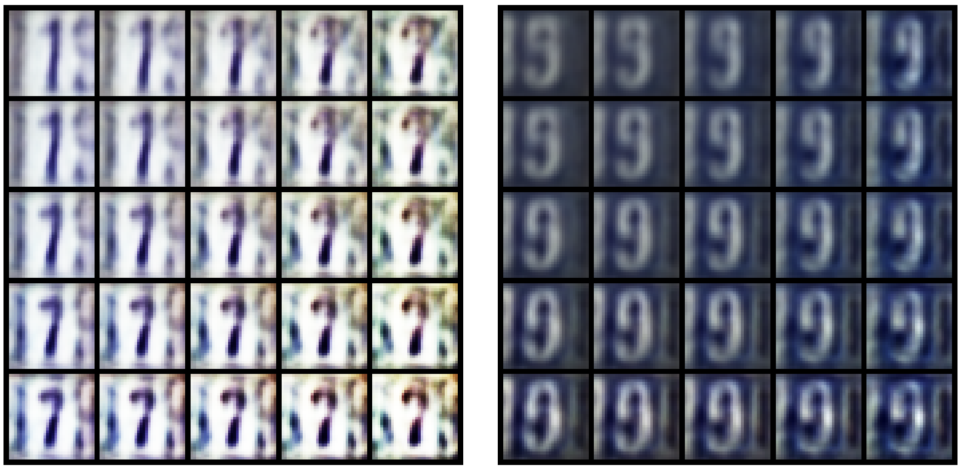
Figure 8: Samples by perturbating the dimensions, left: 65 and 90, right: 8 and 60

3. Interpolation in the data space and in the latent space: Figures 9, 10, 11 and12 show the interpolations in the latent and image space for VAE and GAN respectively. For both models we can observe how the interpolation in the latent space produce images that have more *meaning* compared to the interpolation in the image space, where the transition images are only superposition of the patterns (numbers) in each of the initial images. This can be explained by the latent space forming a kind of *manifold* along which the interpolation makes more sense than in the pixel space.

   (a) VAE:

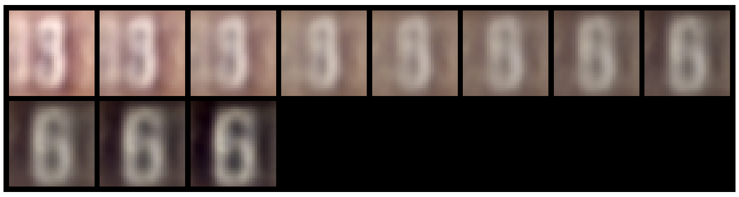
Figure 9: Interpolation in the latent space using VAE

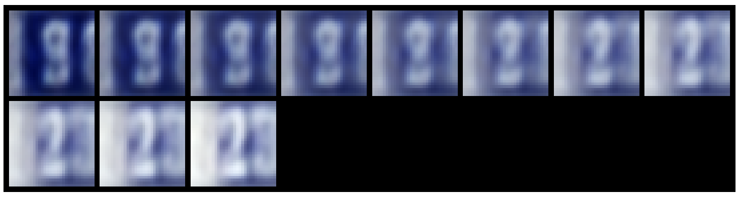
Figure 10: Interpolation in the data space using VAE

   (b) GAN:

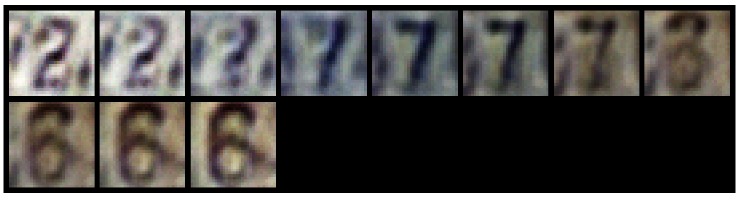
Figure 11: Interpolation in the latent space using GAN

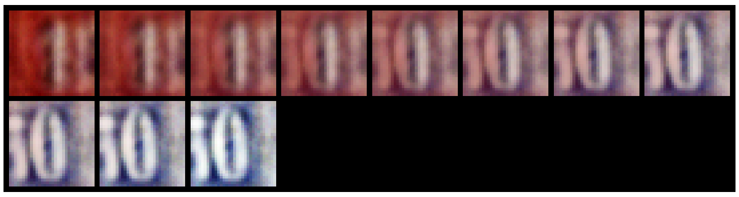
Figure 12: Interpolation in the data space using GAN

B. **Quantitative Evaluations**

1. We have used the provided functions to extract the representations of the images. We compute the Frechet Inception Distance by estimating the mean and covariance of the generator's/decoder's distribution. The calculation steps are explained in the following code snippet:

```
def calculate_fid_score(sample_feature_iterator,
                        testset_feature_iterator):
    gen_features = np.array([])
    test_features = np.array([])

    print("Extracting the features ...")
    #sample_feature_iterator is a generator of a minibatch of features images,
    #that is the last conv2d layer of the classifier of 512 features

    #For generated images
    gen_size=0
    for i in sample_feature_iterator: #iterate over minibatch images
        #Now let's get the activation of the images
        gen_features = np.vstack([gen_features,i.reshape(1,512)])  \
        if gen_features.size else i.reshape(1,512)
        gen_size+=1
        if gen_size==1000: break
    gen_features = gen_features.T

    #For test images
    test_size=0
    for i in testset_feature_iterator: #iterate over test images
        test_features= np.vstack([test_features,i.reshape(1,512)])
        test_size+=1
        if test_size==1000: break
    test_features = test_features.T

    print("Estimating the mean ...")
    #Estimating the mean of the generated images
    mu_gen = np.mean(gen_features,axis=1).reshape(512,1)

    #Estimating the mean of the test images
    mu_test = np.mean(test_features,axis=1).reshape(512,1)

    print("Estimating the variance ...")
    # We use the unbiased variance estimate which
    #is given by (X-mu)(X-mu)^T/(n-1)
    gen_centered  = gen_features - mu_gen
    test_centered = test_features - mu_test

    sigma_gen = np.matmul(gen_centered, gen_centered.T) / (gen_size - 1)
    sigma_test = np.matmul(test_centered, test_centered.T) / (test_size - 1)

    print("Calculating the sqrt of cov matrices product ...")
    # The sqrt of a matrix A needs A to be symetric, but if A, and B
    # are sysmetric A.B is not symeyric necessarly.
    # To solve that we use this trick:
    # sqrt(sigma1 sigma2) = sqrt(A sigma2 A), where A = sqrt(sigma1)
    # the covariance matrix are by definition symetric

    # to prevent negative values in the cov product
    eps = np.eye(512) * 1e-5

    root_sigma_gen = linalg.sqrtm(sigma_gen + eps)
    sigmas_prod = np.matmul(root_sigma_gen,np.matmul(sigma_test, root_sigma_gen))
    # given np.matmul(root_sigma_gen,np.matmul(sigma_test, root_sigma_gen)) is symetric:
    root_sigmas_prod = linalg.sqrtm(sigmas_prod + eps)

    print("Calculating the FID score ...")
    # Calculating the trace
    trace = np.trace(sigma_test + sigma_gen - 2.0 * root_sigmas_prod)

    # Calculate the squared norm between means
    squared_norm = np.sum((mu_test - mu_gen)**2)

    # Calculate the fid score
    fid = squared_norm + trace

    return fid
```

2. We sampled 1000 images from each generative models and calculate the FID-score as instructed. The results are:
   - For the GAN, the FID score is: 29526.37
   - For the VAE, the FID score is: 51355.12

   This metric confirme our ascertainment that the GAN is more realistic than the VAE, given the ground truth given by the provided classifier.

References

[1] Github repository for assignment 3
`https://github.com/faresbs/Representation-Learning.git`