

# 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

1. In this section, we will show the implementation of the Jensen Shannon Divergence (JSD). In order to test this function, we have implemented a neural network with the following architecture:
  - 2 hidden layers, with 64 and 128 output units, respectively, and a ReLu activation function
  - Output layer with Sigmoid activation function

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].

```
1
2 #Implementation of the JSD
3
4 optimizer.zero_grad()
5 p = torch.cat((p1, torch.rand(batch_size, 1)), 1).to(device)
6 q = torch.cat((phi*torch.ones(batch_size, 1), t
7               torch.rand(batch_size, 1)), 1).to(device)
8 Dp = net(p)
9 Dq = net(q)
10 loss = -(math.log(2.) + (1/2.)*torch.mean(torch.log(Dp)) +
11               (1/2.)*torch.mean(torch.log(1-Dq)))
12 loss.backward()
13 optimizer.step()
```

2. In this section, we will present the implementation of the Wasserstein Distance (WD). Same neural network architecture as above was used also in this case with the exception of replacing the Sigmoid output by a linear output.

The following subset of the code shows the implementation of the WD. The full code is also available in our repository [1]

```

1
2 #Implementation of the WD
3
4 optimizer.zero_grad()
5 p = torch.cat((p1,torch.rand(batch_size,1)),1).to(device)
6 q = torch.cat((phi*torch.ones(batch_size,1),
7               torch.rand(batch_size,1)),1).to(device)
8 Dp = net(p)
9 Dq = net(q)
10 # gradient penalty
11 a = torch.rand(batch_size, 1).expand(batch_size,2).to(device)
12 r = a*p + (1-a)*q
13 r.requires_grad = True
14 Dr = net(r)
15 gradients = torch.autograd.grad(outputs=Dr, inputs=r,
16                                grad_outputs=torch.ones(batch_size,1).to(device),
17                                create_graph=True, retain_graph=True, only_inputs=True)[0]
18 loss = -(torch.mean(Dp) - torch.mean(Dq) -
19          gp_coeff*torch.mean((gradients.norm(2, dim=1) - 1) ** 2))
20 loss.backward()
21 optimizer.step()

```

3. Here, we have trained the above neural network with 21 combinations of  $p \sim U(0, Z)$ , and  $q \sim U(\phi, Z)$ , where  $Z \sim U(0, 1)$  and  $\phi$  is a value in the interval  $[-1, 1]$ . The size of the distribution is 512, and the models were trained for 5000 iterations using an SGD optimizer. For every value of  $\phi$  we generate a distribution and measure its distance to  $p \sim U(0, Z)$ . We implement this for WD and JSD and plot their losses.

The full code is given in the file **density\_estimation.py** under our github repository [1].

The following figures show the estimated JSD for the 21 values of  $\phi$  :

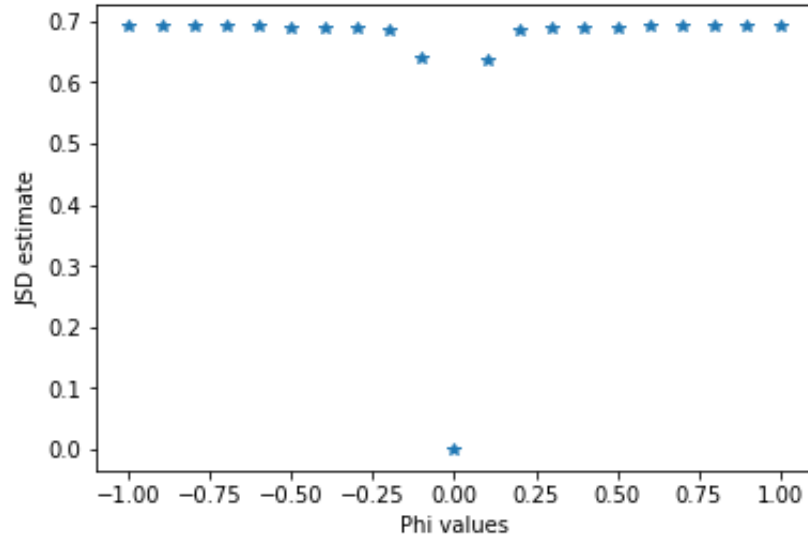


Figure 1: JSD estimation

The estimated WD for the 21 value of  $\phi$  is shown below:

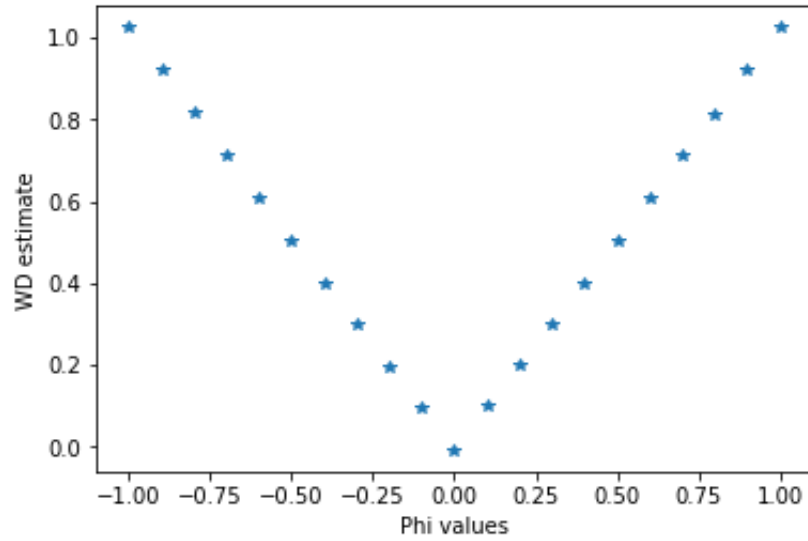


Figure 2: WD estimation

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:

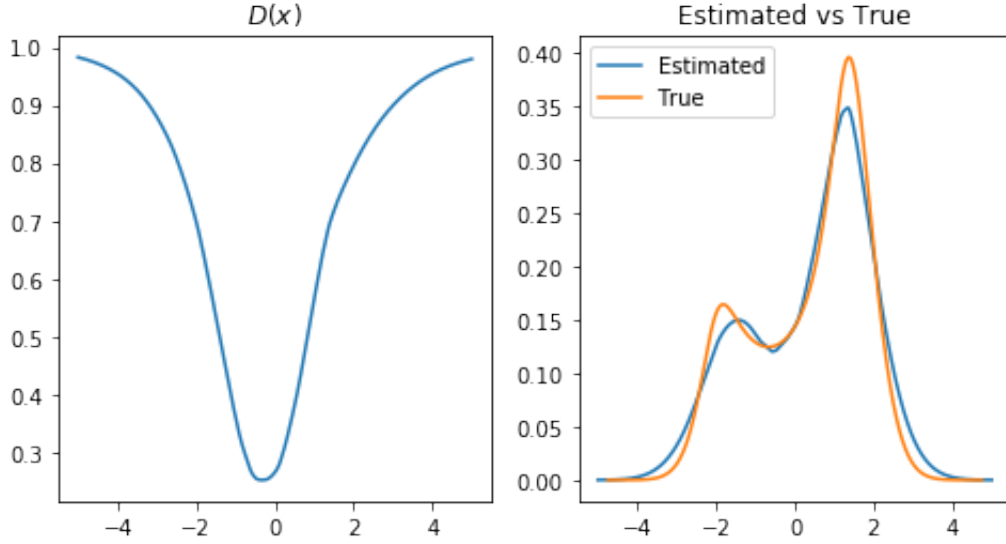


Figure 3: (left) Discriminator output (Right) Estimated  $f_1$

## 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  $-93.58$  on validation set. It is clearly higher than the reference value provided in question.

### 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.

```

1 def loss-IS(model, true_x, z):
2
3     #Loop over the elements i of batch
4     M = true_x.shape[0]
5
6     #Save logp(x)
7     logp_x = np.zeros([M])
8
9     #Get mean and std from encoder
10    #2 Vectors of 100
11    mu, logvar = model.encode(true_x.to(device))

```

```

12 std = torch.exp(0.5*logvar)
13
14 K = 200
15
16 #Loop over tha batch
17 for i in range(M):
18     #z_ik
19     samples = z[i, :, :]
20
21     #Compute the reconstructed x's from sampled z's
22     x = model.decode(samples.to(device))
23
24     #Compute the p(x_i|z_ik) of x sampled from z_ik
25     #Bernoulli dist = Apply BCE
26     #Output an array of losses
27     true_xi = true_x[i, :, :].view(-1, 784)
28     x = x.view(-1, 784)
29
30     p_x = true_xi * torch.log(x) + (1.0 - true_xi) * torch.log(1 - x)
31     p_x = torch.sum(-p_x, dim=1)
32
33     s = std[i, :].view([std.shape[1]])
34     m = mu[i, :].view([std.shape[1]])
35
36     q_z = multivariate_normal.pdf(samples.cpu().numpy(), mean=m.cpu().numpy(),
37                                   cov=np.diag(s.cpu().numpy() ** 2))
38
39     ##p(z_ik) follows a normal dist with mean 0/variance 1
40     #(64, 100)
41     #Normally distributed with loc=0 and scale=1
42     std_1 = torch.ones(samples.shape[1])
43     mu_0 = torch.zeros(samples.shape[1])
44
45     p_z = multivariate_normal.pdf(samples.cpu().numpy(), mean=mu_0.cpu().numpy(),
46                                   cov=np.diag(std_1.cpu().numpy() ** 2))
47
48     #Multiply the probabilities
49     #marginal_likelihood += (p_x * p_z)/q_z
50     #Use logsumexp trick to avoid very small prob
51
52     logp_x[i] = np.log((1.0/K) * np.sum(np.exp(np.log(p_x.cpu().numpy()) + np
53                                           .log(p_z) - np.log(q_z))))
54
55 return logp_x

```

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

- a. Validation: -93.58
- b. Test: -93.63

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

- a. Validation: \* - 93.58
- b. Test: \* - 43.63

Below is a sample of the obtained images generated by the trained model:



Figure 4: A sample of generated images

### 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:

```

1 class Generator(nn.Module):
2     def __init__(self):
3         super(Generator, self).__init__()
4
5         self.model = nn.Sequential(
6             nn.Linear(latent_dim, 128),
7             nn.ReLU(),
8             nn.Linear(128, 256),
9             nn.ReLU(),
10            nn.Linear(256, 512),
11            nn.ReLU(),
12            nn.Linear(512, 1024),
13            nn.ReLU(),
14            nn.Linear(1024, 2048),
15            nn.ReLU(),
16            nn.Linear(2048, int(np.prod(img_shape))),
17            nn.Tanh()
18        )
19
20    def forward(self, z):
21        img = self.model(z)
22        img = img.view(img.shape[0], *img_shape)
23        return img

```

We have decided to go with this architecture, after having tested several architectures for both models, including convolutional 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.

#### A. Qualitative Evaluations

1. Visual samples We have generated different samples from both models as we can see below in Figures 5 and 6. We notice that the images generated by the VAE model are very clear but a little blurry, whereas the images generated by the GAN model are more diversified, and seem more realistic.



Figure 5: Samples generated with VAE

@FARRIS PUT HERE SOME SAMPLES



Figure 6: Samples generated with GAN.

2. Learning the disentangled representation in the latent space The following figures show how the GAN has learned a disentagled representation:

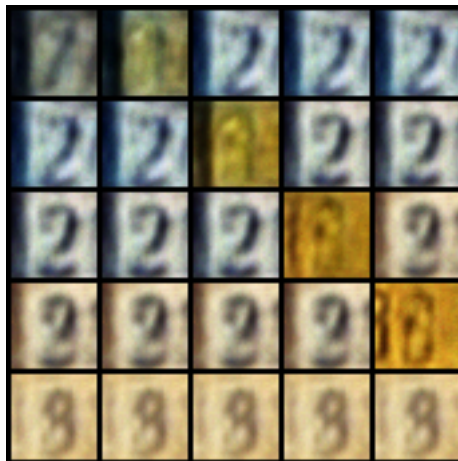


Figure 7: Samples by perturbing dimensions 6 and 96





Figure 8: Samples by perturbing dimensions 25 and 75

### TO BE CONTINUED BY FARRIS FOR VAE

3. Interpolation in the data space and in the latent space:

TBD BY JORGE FOR GAN AND FARRIS FOR VAE

### 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:

```

1 def calculate_fid_score(sample_feature_iterator,
2                         testset_feature_iterator):
3     gen_features = np.array([])
4     test_features = np.array([])
5
6     print("Extracting the features ...")
7     #sample_feature_iterator is a generator of a minibatch of features
8     #images,
9     #that is the last conv2d layer of the classifier of 512 features
10
11     #For generated images
12     gen_size=0
13     for i in sample_feature_iterator: #iterate over minibatch images
14         #Now let's get the activation of the images
15         gen_features = np.vstack([gen_features, i.reshape(1,512)]) \
16         if gen_features.size else i.reshape(1,512)
17         gen_size+=1
18         if gen_size==1000: break
19     gen_features = gen_features.T
20
21     #For test images
22     test_size=0
23     for i in testset_feature_iterator: #iterate over test images
24         test_features= np.vstack([test_features, i.reshape(1,512)])
25         test_size+=1
26         if test_size==1000: break

```

```

26 test_features = test_features.T
27
28 print("Estimating the mean ...")
29 #Estimating the mean of the generated images
30 mu_gen = np.mean(gen_features , axis=1).reshape(512,1)
31
32 #Estimating the mean of the test images
33 mu_test = np.mean(test_features , axis=1).reshape(512,1)
34
35 print("Estimating the variance ...")
36 # We use the unbiased variance estimate which
37 #is given by  $(X-\mu)(X-\mu)^T/(n-1)$ 
38 gen_centered = gen_features - mu_gen
39 test_centered = test_features - mu_test
40
41 sigma_gen = np.matmul(gen_centered , gen_centered.T) / (gen_size - 1)
42 sigma_test = np.matmul(test_centered , test_centered.T) / (test_size
43 - 1)
44
45 print("Calculating the sqrt of cov matrices product ...")
46 # The sqrt of a matrix A needs A to be symmetric , but if A, and B
47 # are symmetric A.B is not symmetric necessarily.
48 # To solve that we use this trick:
49 #  $\sqrt{\sigma_1 \sigma_2} = \sqrt{A \sigma_2 A}$ , where  $A = \sqrt{\sigma_1}$ 
50 # the covariance matrix are by definition symmetric
51
52 # to prevent negative values in the cov product
53 eps = np.eye(512) * 1e-5
54
55 root_sigma_gen = linalg.sqrtm(sigma_gen + eps)
56 sigmas_prod = np.matmul(root_sigma_gen , np.matmul(sigma_test ,
57 root_sigma_gen))
58 # given  $\text{np.matmul}(\text{root\_sigma\_gen}, \text{np.matmul}(\text{sigma\_test}, \text{root\_sigma\_gen}))$  is symmetric:
59 root_sigmas_prod = linalg.sqrtm(sigmas_prod + eps)
60
61 print("Calculating the FID score ...")
62 # Calculating the trace
63 trace = np.trace(sigma_test + sigma_gen - 2.0 * root_sigmas_prod)
64
65 # Calculate the squared norm between means
66 squared_norm = np.sum((mu_test - mu_gen)**2)
67
68 # Calculate the fid score
69 fid = squared_norm + trace

```

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 confirms 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>