Assignment 3
Deep Generative Models

Francesco Dal Canton
f.dal.canton@student.uva.nl
12404225

1 Variational Auto Encoders

1.1 Latent Variable Models

1.1
1. VAEs and VAs are different in terms of their main function.
   VAs are used for dimensionality compression: the encoder and the decoder are part of a single network which approximates the identity function. Since the network as a bottleneck between encoder and decoder, the activations of that bottleneck serve as reduced features for the input. While the decoder part of the network may be used for generation, by providing some random features for the bottleneck layer, we can't sample from the distribution of the data.
   A VAE works differently, since it is a generative model. VAEs still use a bottleneck structure, but in this case the encoder learns the data's probability distribution, which makes it possible to sample from said distribution and generate an output similar to the input sample. In that sense, the decoder of a VAE works similarly as to that of a VA.
2. This was answered in the previous question. Since VAs don't model the data's distribution (be it mapped to a latent space or not), they are not generative models.
3. One of the main purposes of AEs is to produce a reduced dimensionality version of the input sample. This means that the encoder should be able to map each sample to a tensor of reduced dimensionality. Since VAEs model a distrubution over the data instead of specific latent representations for single samples, it might be possible to use them for data compression if we take the distribution parameters to be the features of the compressed input.
4. This was answered in the first question. VAEs are generative since they explicitly model the distribution of the data (or, rather, the distribution of the latent variables), while VAs don't.

1.2 Decoder: The Generative Part of the VAE

1.2 The procedure used to sample from such a model is, as mentioned, ancestral sampling. This technique may be applied to any graphical model expressing variable dependencies. The idea is that we first sample from variables that do not depend on others, and once we have sampled those, we may sample the ones that depend on them. This can be repeated iteratively until we have sampled all variables in the model.

In our case we only have two variables $\boldsymbol{z}_n$ and $\boldsymbol{x}_n$, where the latter depends on the former. This means that we first sample $\boldsymbol{z}_n \sim \mathcal{N}(0, \boldsymbol{I}_D)$, which allows us to use the sampled $\boldsymbol{z}_n$ in order to sample $p(\boldsymbol{x}_n|\boldsymbol{z}_n)$ according to equation:

$$p(\boldsymbol{x}_n|\boldsymbol{z}_n) = \prod_{m=1}^{M} \text{Bern}\left(\boldsymbol{x}_n^{(m)}|f_\theta(\boldsymbol{z}_n)_m\right)$$

1.3 The key element that makes it so that the assumption is not restrictive is the term $f_\theta(\boldsymbol{z}_n)_m$ in the equation above. This term indicates that, when computing $p(\boldsymbol{x}_n|\boldsymbol{z}_n)$, we are actually using a parametric function of $\boldsymbol{z}_n$ as its input. Carl Doersch's tutorial reminds us that if this function is complex enough, we can map a normally distributed $p(\boldsymbol{z}_n)$ to any n-dimensional distribution. We can assume that the VAE's decoder's first few layers will perform exactly this operation, mapping the latent variables to the actual distribution of the data before it is reconstructed.

1.4 (a) We can approximate $\log p(\boldsymbol{x}_n)$ using Monte-Carlo Integration as follows:

$$
\begin{aligned}
\log p(\mathcal{D}) &= \sum_{n=1}^{N} \log \int \log p(\boldsymbol{x}_n|\boldsymbol{z}_n) p(\boldsymbol{z}_n) d\boldsymbol{z}_n \\
&\approx \sum_{n=1}^{N} \log \frac{1}{N} \sum_{i=1}^{S} p\left[\boldsymbol{x}_n|\boldsymbol{z}_n^{(i)}\right] \quad \text{with} \quad \boldsymbol{z}_n^{(i)} \sim p(\boldsymbol{z}_n) \\
&\approx \sum_{n=1}^{N} \log \frac{1}{N} \sum_{i=1}^{S} \prod_{m=1}^{M} \text{Bern}\left(\boldsymbol{x}_n^{(m)}|f_\theta\left[\boldsymbol{z}_n^{(i)}\right]_m\right)
\end{aligned}
$$

Where S is the total number of samples we will take when computing the approximation.

(b) The inefficiency problem that this method brings boils down to the issue of sparsity in high-dimensional spaces. In order to approximate $\log p(\boldsymbol{x}_n)$, we sample a number of $\boldsymbol{z}_n^{(i)} \sim p(\boldsymbol{z}_n)$. As the dimensionality of $\boldsymbol{z}$ increases, however, the search space increases exponentially. This in turn means that we need many more samples to properly approximate $\log p(\boldsymbol{x}_n)$, making the procedure inefficient.
This problem is made worse by the presence of the $p(\boldsymbol{x}_n|\boldsymbol{z}_n)$ in the integral above. Even if we sample $\boldsymbol{z}_n^{(i)}$, it might be that $p(\boldsymbol{x}_n|\boldsymbol{z}_n)$ is incredibly small given the sparsity of the high-dimensional space. This means that the calculation will have almost no impact on the learning procedure, and it will have the only effect of making it slower.

1.3 The Encoder: $q_\phi(z_n|x_n)$

1.5 (a) The value of the KL-divergence for two distributions is high if the two distributions are different. For Gaussian distributions, this boils down to having a high $|\mu_q - \mu_p|$ and $|\sigma_q^2 - \sigma_p^2|$. As such, a small KL-divergence would be given by, for instance, $q^- = \mathcal{N}(0.1, 0.9)$. A high KL-divergence would be given by any sufficiently different distribution, such as $q^+ = \mathcal{N}(64, 512)$.

(b) The closed-form formula for $D_{KL}(q\,||\,p)$ is:

$$
D_{KL}(q\,||\,p) = \log \frac{\sigma_p}{\sigma_q} + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} - \frac{1}{2}
$$

With this we can also check the answer to (a). In fact, $D_{KL}(q^-\,||\,p) \approx 0.154$, while $D_{KL}(q^+\,||\,p) \approx 133,113.261$.

1.6 The reason why the right hand side of that equation is called the *lower bound* is because the log-probability will never be lower than it. To see this, consider that $D_{KL}(r\,||\,s) \geq 0$ for any two r and s distributions. As such, we can compress the second term of the left hand size into arbitrary positive constant c. This way we have that:

$$
\begin{aligned}
\log p(\boldsymbol{x}_n) - c &= \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_n)}\left[\log p(\boldsymbol{x}_n|Z)\right] - D_{KL}\left[q(Z|\boldsymbol{x}_n)\,||\,p(Z)\right] \\
\implies \log p(\boldsymbol{x}_n) &\geq \mathbb{E}_{q_\phi(\boldsymbol{z}|\boldsymbol{x}_n)}\left[\log p(\boldsymbol{x}_n|Z)\right] - D_{KL}\left[q(Z|\boldsymbol{x}_n)\,||\,p(Z)\right]
\end{aligned}
$$

1.7 Optimizing the log-probability $\log p(\boldsymbol{x}_n)$ involves an intractable integral, which we can't successfully approximate even by using Monte-Carlo Integration. Furthermore, we can't even compute the error term given by $D_{KL}\left[q(Z|\boldsymbol{x}_n)\,||\,p(Z|\boldsymbol{x}_n)\right]$, since we have no way of computing $p(Z|\boldsymbol{x}_n)$.

1.8 Since the left-hand side of the equation has two terms, when pushing up the lower bound only two things can happen. The first is that we are actually increasing the log-probability of the data. The second is that we are more accurately modelling the probability of the latent variables given our data (i.e. $q(Z \,||\, \boldsymbol{x}_n)$ is closer to $p(Z|\boldsymbol{x}_n)$).

1.4 Specifying the encoder $q_\phi(z_n|x_n)$

1.9 The expectation of the log-probability of the input given our latent variables may be seen as a *reconstruction* term because of the role it plays in the VAE. That is, we approximate this term by sampling from the latent distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x}_n)$, which is modeled by the encoder part of our model. The act of sampling involves feeding our input to the encoder, but the distribution that we are trying to model lies in the decoder, which *reconstructs* the input based on the sampled distribution. In that sense, since our decoder directly models $p_\theta(\boldsymbol{x}_n|Z)$ and is reconstructing the input, this is an appropriate name for the loss.

On the other hand, the second term acts as a *regularization* loss because it makes sure that the latent variable distribution we are modelling doesn't deviate too much from $\mathcal{N}(0, I)$. This has a similar effect to what the L^2 regularization or weight decay do for neural networks. By forcing the distribution to remain close to the unit, it reduces overfitting by making sure the model doesn't populate vastly distant areas of the latent space and overfits on the data.

1.10 We consider the case where we compute the loss after sampling $\boldsymbol{z}$ only once. Therefore, by sampling $\boldsymbol{z}_n \sim q_\phi(\boldsymbol{z}|\boldsymbol{x}_n)$, we have:

$$
\begin{aligned}
\mathcal{L}_n^{recon} &\approx -\log p_\theta(\boldsymbol{x}_n|\boldsymbol{z}_n) \\
&\approx -\log\left[\prod_{m=1}^{M} \text{Bern}\left(\boldsymbol{x}_n^{(m)}|f_\theta\left[\boldsymbol{z}_n^{(i)}\right]_m\right)\right] \\
&\approx -\sum_{m=1}^{M} \log\left[\left(f_\theta\left[\boldsymbol{z}_n^{(i)}\right]_m\right)^{\boldsymbol{x}_n^{(m)}}\left(1-f_\theta\left[\boldsymbol{z}_n^{(i)}\right]_m\right)^{(1-\boldsymbol{x}_n^{(m)})}\right] \\
&\approx -\sum_{m=1}^{M}\left[\boldsymbol{x}_n^{(m)}\log f_\theta\left(\boldsymbol{z}_n^{(i)}\right)_m + \left(1-\boldsymbol{x}_n^{(m)}\right)\log\left(1-f_\theta\left[\boldsymbol{z}_n^{(i)}\right]_m\right)\right]
\end{aligned}
$$

As for the regularization term, Carl Doersch's tutorial gives us the answer:

$$
\mathcal{L}_n^{reg} = \frac{1}{2}\left[\text{tr}\left[\Sigma_q(\boldsymbol{x}_n)\right] + \mu_q(\boldsymbol{x}_n)^\top \mu_q(\boldsymbol{x}_n) - D - \log|\Sigma_q(\boldsymbol{x}_n)|\right]
$$

Where D is the dimensionality of the distribution, and $|A|$ is the determinant of matrix A.

This gives us a closed-form expression for the loss given one sample.

1.5 The Reparametrization Trick

1.11 (a) We're using gradient descent to optimize the lower bound of the log-likelihood, and the closed-form expression for that lower bound is given by $\mathcal{L}$. As such, we need to take its gradient.

(b) The problem of computing $\nabla_\phi \mathcal{L}$ boils down to the presence of a non-continuous operation in the expression. That is, we can't compute the gradient for the operation of sampling $\boldsymbol{z}_n \sim q_\phi(\boldsymbol{z}|\boldsymbol{x}_n)$.

(c) The *reparametrization trick* is shown very clearly in Figure 4 from Carl Doersch's tutorial. It involves separating the operation of sampling from the rest by assigning it to an input layer, which samples $\epsilon \sim \mathcal{N}(0, I)$. This way, $\mu_q(\boldsymbol{x}_n)$ and $\Sigma_q(\boldsymbol{x}_n)$ can be computed by the encoder, after which we can compute $\boldsymbol{z}_n = \mu_q(\boldsymbol{x}_n) + \epsilon * \Sigma_q^{\frac{1}{2}}(\boldsymbol{x}_n)$. With this we can take the sampling operation out of the gradient when optimizing.

1.6 Putting things together: Building a VAE

1.12 The implementation of the VAE was straightforward and followed the provided template. Aside for the obvious details that pertain the model itself, I made a few design choices.

i. The prediction of the mean and variance for the latent variable distribution was done by two separate layers, which took the same input (i.e. the output of the first hidden layer of the encoder).
ii. The layer computing the variance used a ReLU nonlinearity to constrain its output to positive values.
iii. In computing the losses, I added a small constant when computing logarithms, in order to avoid undefined results.
iv. For each experiment I trained the model for 40 epochs.

1.13 The plot of the estimated lower-bounds (ELBO) for the 40 epochs of training using a 20-dimensional latent space are shown in Figure 1.

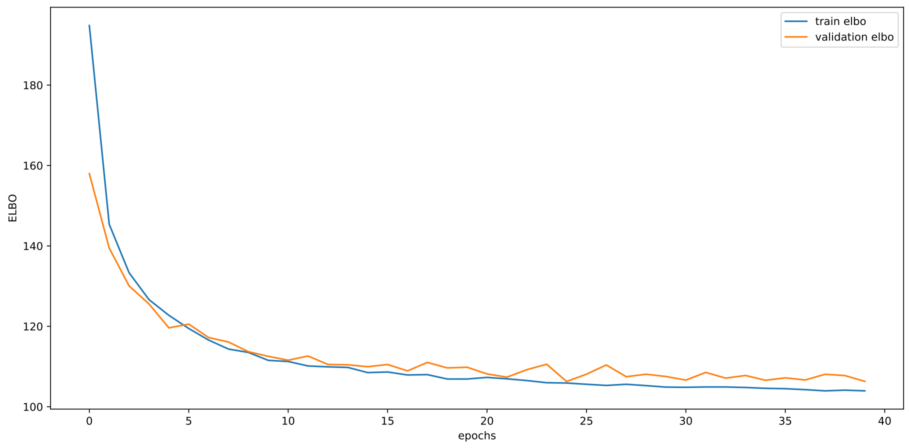

Figure 1: Plot of the training and evaluation ELBO curves throughout the 40 epochs of training.

1.14 Here I plot 25 samples from the model at three equidistant moments during training: on the first epoch, on the 20th epoch, and on the last epoch. The samples are shown in Figure 2.

1.15 The 2-dimensional manifold for the VAE trained for 40 epochs is shown in Figure 3.

2 Generative Adversarial Networks

2.1 The generator takes as input a vector z sampled from some noise distribution. It outputs, in our case, an image of the same size and value range as the ones in the training set.

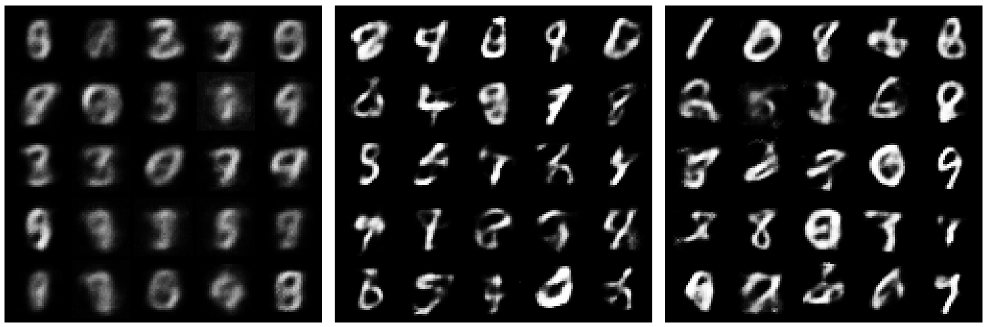

Figure 2: 25 samples from the VAE at different points during the training: from left to right at epoch 1, 20, and 40.

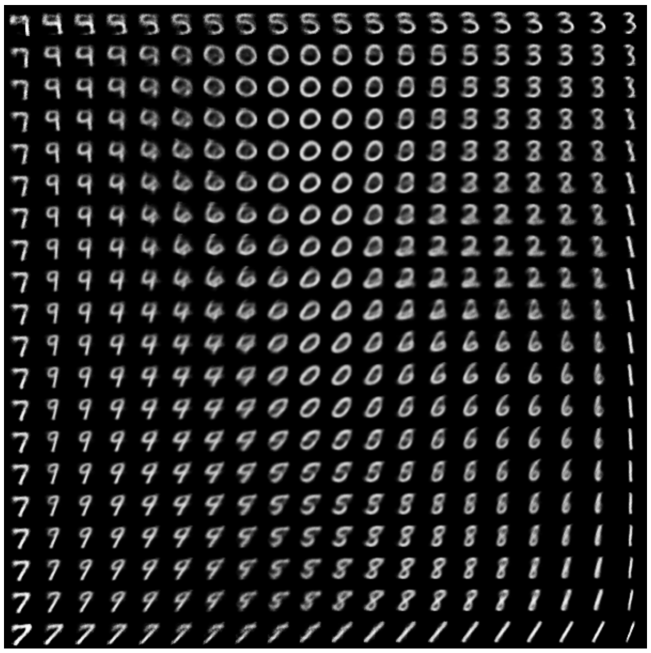
Figure 3: Two-dimensional manifold for the VAE trained for 40 epochs.

The discriminator takes as input an image of the same type as the output of the generator (which may be the output of the generator or an actual image from the training set), and returns a score indicating how confident it is about the image being real or generated.

2.1 Training objective: A Minimax Game

2.2 The first term, $\mathbb{E}_{p_{data}(x)}\left[\log D(X)\right]$, refers to the expectation of the log-likelihood that the discriminator will successfully identify real pictures, given the distribution of the data.

The second term, $\mathbb{E}_{p_z(z)}\left[\log(1 - D[G(Z)])\right]$ refers to the expectation of the log-likelihood that the discriminator will successfully identify the generated images, given the noise distribution that they are being generated from.

2.3 From the perspective of game theoretical perspective, in the ideal scenario after convergence the generator and discriminator should have reached the Nash equilibrium. Since this is a MiniMax game, we know that this occurs when the two parties win equally as likely, meaning that $D(X) = \frac{1}{2}$ and $D[G(Z)] = \frac{1}{2}$

This conclusion would be reached if the two models were trained in such a way that they both reach peak performance, at which point both win equally as often.

2.4 The problem with the second term arises from the fact that the discriminator is much easier to train compared to the generator. As such, it may happen that the second term saturates quickly, leaving a gradient of 0 so that the generator can't learn anymore. At that point, the generator's performance plateaus.

A solution is to use a different loss function for the generator. Namely, we can use $\log D(G[Z])$, which should be maximized instead of being minimized. Using this, the generator receives a strong gradient during backpropagation even if the discriminator has a perfect score (i.e. $\lim_{x \to 0} \frac{\partial \log x}{\partial x} = \infty$)

2.2 Building a GAN

2.5 I did not have the time to implement the GAN.

2.6

2.7

3 Generative Normalizing Flows

3.1 Change of variables for Neural Networks

3.1 For the multivariate case, the only two changes to apply are the use of vector notation, and the use of the absolute value of the determinant of the Jacobian instead of that of the derivatives themselves. As such, for eq. 16 we have:

$$
\begin{aligned}
\vec{z} &= f(\vec{x}) \\
\vec{x} &= f^{-1}(\vec{z}) \\
p(\vec{x}) &= p(\vec{z}) \left| \det \frac{\partial \vec{f}}{\partial \vec{x}} \right|
\end{aligned}
$$

While for eq. 17 we have:

$$
\log p(\vec{x}) = \log p(\vec{z}) + \sum_{l=1}^{L} \log \left| \det \frac{\partial \vec{h}_l}{\partial \vec{h}_{l-1}} \right|
$$

3.2 The first criterion is that $f(\vec{x})$ needs to be invertible. The second is that the Jacobian matrix should be square, otherwise we can't compute its determinant. This second constraint implies that the dimensionality of $\vec{x}$ and $\vec{z}$ should be the same. Furthermore, the determinant of the jacobian should be non-zero, since that would make the last logarithm undefined.

3.3 The problem with optimizing the aforementioned equation is that both computing the Jacobian and computing the determinant are very computationally expensive. Computing the Jacobian is in $\mathcal{O}(n^3)$. Computing the determinant is also expensive but its computational complexity varies depending on the Jacobian's properties.

3.4 The consequence of this would be an uneven distribution mapping. In other words, since the pixel values are discrete, we don't have a complete distribution. When mapping the initial (quantized) distribution through the various normalizing flows, this will produce uneven and sparse intermediate and final distributions, so that it will be very hard to sample from them.

In order to fix this problem, the paper cited in the question proposes among others a technique called *uniform quantization*. For an input image where each of the D pixels take an integer value in the range $(0, 255)$, this means adding $\epsilon \sim \mathcal{U}[0, 1)^D$ to the input. The effect is to fill the spaces inbetween discrete pixel values, this producing a smoother distribution.

3.2 Building a flow-based model

3.5 During training the flow-based model takes as input a data point from the dataset (e.g. an image). The output, after the sample has been mapped through the model's flows, should be a data point of the same dimensionality but belonging to a different distribution, one that allows for easier sampling.

During inference time the process is reversed: we feed the model a data point sampled from a noise distribution of the same dimensionality as the training examples, and we obtain as output an image of that same dimensionality that resembles the ones from our training dataset. Note that during inference the flow is reversed compared to the training configuration.

3.6 During training we want to iteratively perform these steps:

1. Feed samples from our dataset into the flow-based model, and receive a data point from the output distribution.
2. Since we computed the determinants of the Jacobian for each pair of consecutive flows during forward propagation, we use those instead of computing them again. We plug them into our loss function (i.e. the negative log-likelihood) along with the log-probability of the output data point. We can now backpropagate the loss through our model.

During inference we will do the opposite:

1. Feed a data point sampled from the prior distribution into the reverse of the flow.
2. As output we obtain a data point similar to the data points in our original dataset.

3.7 I did not have the time to implement the GNF.

3.8

4 Conclusion

4.1 In this assignment we explored three generative models, which approach the task of approximating the log-likelihood of the data in different ways.

Variational Auto Encoders do this by optimizing a quantity that can be brought back to the log-likelihood of the data: the ELBO. By modelling a latent variable to encode the data, we can maximize the ELBO and train a generator for new datapoints. Hence, the log-likelihood of the data is modelled indirectly by manipulating distributions and their divergences.

Generative Adversarial Networks, on the other hand, don't model the log-likelihood of the data at all. This configuration involves a generative network and a discriminative network. The two compete in a zero-sum MiniMax game, in which the generator learns to mimic the real data as closely as possible by making the discriminator classify its output as real data.

Generative Normalizing Flows are the only one of the three models that directly maximizes the log-likelihood of the data. By mapping the original data's complex distribution through a series of invertible functions, this model simplifies the distribution (effectively mapping it to noise). By doing this, it makes it possible to sample from the noise distribution and feed the sampled data point through the reverse model, in order to generate novel samples.