Assignment 3: Deep Generative Models

Alexandra Lindt
alex.lindt@protonmail.com

1 Variational Auto Encoders

1.1 Latent Variable Models

This first part of the assignment deals with the Variational Autoencoder by Kingma and Welling [2013]. Please note that all given answers are based on information from the original paper as well as on the tutorial by Doersch [2016].

Question 1.1

When comparing the Variational Autoencoder with the standard Autoencoder, the first thing to notice is that both models have the same overall structure: The input is transformed to an internal or *latent* representation by an encoder network g and following reconstructed by a decoder network f. However, besides this structural commonality the two models have a lot of differences.

1. and 4.) When training an standard Autoencoder, one is typically interested in obtaining an efficient encoding of a data set. The model is trained to optimize a loss function

$$L_{AE} = L(X, f_\theta(g_\psi(X)) \ ,$$

where X denotes an training data sample and L denotes a function that measures the difference between its two inputs (e.g. Euclidean Distance). By optimizing the model to resemble the input sample as close as possible in its output, it is forced to represent the input as good as possible with its internal latent variable. Since the encoder f_θ and the decoder g_ψ represent deterministic functions, the latent representation for a specific input X is one fixed z. Therefore, after training, the latent variable can be used as an expressive representation for the respective data sample. There further exist variants of the Autoencoder that extend the loss function with a term that restricts the choice of latent variables, depending on what kind of representation for the data is desired (e.g. sparse, disentangled, ..).

In contrast, the Variational Autoencoder maximizes the function

$$\begin{aligned} L_{VAE} &= \mathbb{E}_{X\sim D}[\mathbb{E}_{z\sim Q}[\log(f_\theta(z)) - D_{KL}(g_\psi(X), P(z))] \ ] \\ &= \mathbb{E}_{X\sim D}[\mathbb{E}_{z\sim Q}[\log(P(X|z)) - D_{KL}(Q(z|X), P(z))] \ ] \end{aligned}$$

As already apparent from this function, encoder g_ψ and decoder f_θ are not defined by deterministic functions but by probability distributions. In the case of the encoder f_θ, this is because it does not output a latent variable z directly, but instead two parameter vectors that describe a probability distribution. The decoder g_ψ still outputs a single output directly. However, it is trained on constructing it from not only one specific z but also to merely approach it from neighbouring latent variables. As a consequence, for a given input z the decoder output can be regarded a probability distribution over X. Regarding the concrete loss-function given above, the first term is the reconstruction log-likelihood. Like in standard autoencoders, it encourages the decoder to reconstruct X as good as possible and the encoder to output a probability distribution that describes X as well as possible. The second loss function term is the Kullback-Leibler divergence between the encoder's distribution $f_\theta(X) = Q(z|X)$ and the prior distribution over z, $P(z)$. This term is minimized if $Q(z|X) = P(z)$, which leads to the encoder outputting latent vectors z according to the prior distribution. As the prior is assumed to be the Gaussian distribution $\mathcal{N}(0, I_D)$, the encoder generally outputs zs that are close in latent

space. Consequently, when sampling a random vector from the prior distribution after training, the vector cannot fall into a huge gap between samples that the decoder network was trained to decode. Therefore, the decoder is able to synthesize a realistic output for such z by smoothly mixing the characteristics of close latent vectors it has been trained on.

2.) In contrast to the Variational Autoencoder, standard Autoencoders that are trained for reconstruction are no generative models. As already described above, this is because their decoder networks learns to reconstruct a specific X for a specific z. Therefore, it is likely that the encoder network will choose very dissimilar latent representations during training, to ensure that they are not mistaken for each other by the decoder network. This leads to the decoder network not learning a dense mapping from the prior distribution $P(z)$ to the image space and therefore no new images can be sampled.

3.) One could use a Variational Autoencoder instead of a standard Autoencoder. Even if the overall z are close in latent space, a z generated from the encoder for an image X is still representative for X and can be reconstructed by the decoder network into an image that resembles X.

1.2 Decoder: The Generative Part of the VAE

Question 1.2

As given in this Standford Lecture (Ermon [2018-19]), *ancestral sampling* or *forward sampling* means to sample random variables in topological order. First, variables with no parent variables are sampled and following the dependent variables are sampled conditioned on them. In case of the Variational Autoencoder, the random variables with no parent variables are the latent variables z and the output random variables X (i.e. newly generated images) are conditioned on z via the deterministic decoder network f_θ. For sampling a new data point, we therefore proceed as follows:

1. We sample a $z_n \in \mathbb{R}^D$ from its assumed prior distribution.
$$z_n \sim \mathcal{N}(0, I_D)$$
2. We sample a $x_n \in \mathbb{R}^M$ by passing the sampled z_n through the decoder network.
$$x_n = f_\theta(z_n)$$

Question 1.3

As we can see in the given equation

$$p\left(x_n|z_n\right) = \prod_{m=1}^{M} \text{Bern}\left(x_n^{(m)}|f_\theta\left(z_n\right)_m\right)$$

the probability $p\left(x_n|z_n\right)$ depends on the output of a latent neural network f (i.e. the decoder network) with parameters θ for an input z_n. As already stated in the task description, Variational Autoencoders assume that the z_n in the above equation is drawn from the normal distribution $\mathcal{N}(0, I_D)$. To understand why this assumption is not very restrictive in the given context, we have to consider that every D-dimensional distribution can be generated by applying a sufficiently complicated function on D normally distributed variables (Doersch [2016]). Since the function f_θ is represented by a sufficiently deep neural network, we can think of the first few layers as a mapping from a normally distributed z_n to its true latent representation and of the following layers a mapping from latent representation to the output. Generally, it does not matter whether this separation between both tasks in f_θ is actually happening. If a specific latent structure leads to the network generating more accurate outputs, we can be sure that the network will represent the mapping from a normally distributed input z_n to this latent structure with some of its layers.

Question 1.4

(a) Looking at the hinted equation

$$\log p(\mathcal{D}) = \sum_{n=1}^{N} \log \mathbb{E}_{p(z_n)}\left[p\left(x_n|z_n\right)\right]$$

we can see that we actually can approximate $\log p(\mathcal{D})$ with Monte Carlo sampling by calculating for each x_n

$$\mathbb{E}_{p(z_n)} p\left(x_n|z_n\right) = \int_{z_n} p(x_n|z_n)p(z_n) \approx \frac{1}{K}\sum_{k=1}^{I} p(x_n|z_k) \quad \text{with} \quad z_k \sim p(z_n)$$

for a large number K of sampled z.

(b) For making this estimate approximately correct, we need to cover as much as possible from the latent space with the sampled z_n to get a good approximation for the Monte Carlo intergral. Therefore, the number of samples K needs to be extremely high. However, for high dimensional latent spaces, this becomes infeasible. Another problem is that most of the sampled z_n will contribute nothing to our estimate, because the corresponding probability $p(x_n|z_n)$ is close to zero. It would therefore be desirable to know in which part of the latent space to sample, in order to get z_n that result in a useful (i.e. non-zero) $f_\theta(z_n) = p(x_n|z_n)$.

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

Question 1.5

(b) As stated in Doersch [2016], the closed form for the KL divergence for two Gaussian distributions $q(x) = \mathcal{N}(\mu_1, \Sigma_1)$ and $p(x) = \mathcal{N}(\mu_2, \Sigma_2)$ is given as

$$D_{\mathrm{KL}}(q\|p) = \frac{1}{2}\Bigg(\operatorname{tr}(\Sigma_2^{-1}\Sigma_1 - I) + (\mu_1 - \mu_2)^T\Sigma_2^{-1}(\mu_1 - \mu_2) + log(\frac{|\Sigma_2|}{|\Sigma_1|})\Bigg) \ .$$

Assuming p and q are two univariate Gaussian distributions with $q(x) = \mathcal{N}(\mu_1, \sigma_1^2)$ and $p(x) = \mathcal{N}(\mu_2, \sigma_2^2)$, we can rewrite this as

$$D_{\mathrm{KL}}(q\|p) = \frac{1}{2}\Bigg(\frac{\sigma_1^2}{\sigma_2^2} - 1 + \frac{1}{\sigma_2^2}(\mu_1 - \mu_2)^2 + log(\frac{\sigma_2^2}{\sigma_1^2})\Bigg) \ ,$$

and when inserting the given $p(x) = \mathcal{N}(\mu_2, \sigma_2) = \mathcal{N}(0, 1)$, we receive

$$D_{\mathrm{KL}}(q\|p) = \frac{1}{2}\Bigg(\sigma_1^2 - 1 + (\mu_1)^2 - log\big(\sigma_1^2\big)\Bigg).$$

(a) As the KL divergence is defined to be ≥ 0, it is minimal when it's 0. That happens if $\mu_q = 0$ and $\sigma_q^2 = 1$, i.e. if $q(x) = p(x)$. In this case, we receive

$$D_{\mathrm{KL}}(q\|p) = \frac{1}{2}\Bigg(1 - 1 + 0 - log(1)\Bigg) = 0$$

From the above formula, we see that $D_{\mathrm{KL}}(q\|p)$ is higher the higher the absolute value of μ_1 is. Further, if we take a look at $\sigma_1^2 - log\big(\sigma_1^2\big)$, we can see that this term gets higher for high σ_1^2 or for σ_1^2 close to 0. Therefore, an example case for getting a really high KL divergence here would be $q(x) = \mathcal{N}(10.000, 1.000)$:

$$D_{\mathrm{KL}}(q\|p) = \frac{1}{2}\Bigg(1.000 - 1 + 10.000^2 - log(1.000)\Bigg) = 50000496.046$$

Question 1.6

When looking at the given equation 11

$$\log p(x_n) - D_{\mathrm{KL}}(q(x_n)||p(Z|x_n)) = \mathbb{E}_{q(z|x_n)}[\log p(x_n|Z)] - D_{\mathrm{KL}}(q(Z|x_n)\|p(Z))$$

and knowing from the previous task that the Kullback–Leibler divergence is defined as ≥ 0 , we can see that

$$\log p(x_n) - D_{\mathrm{KL}}(q(x_n)||p(Z|x_n)) \leq \log p(x_n)$$

and hence

$$\mathbb{E}_{q(z|x_n)}[\log p(x_n|Z)] - D_{\mathrm{KL}}(q(Z|x_n)\|p(Z)) \leq \log p(x_n) \ .$$

We can therefore consider the right-hand side of equation 11 a lower bound of the log-probability $\log p(x_n)$.

Question 1.7

The actual log-probability is defined as an integral over all latent representations

$$\log p(x_n) = \log \int_z p(x_n, z) dz$$

which is generally intractable as I already mentioned above. We therefore optimize the log-probability's lower bound

$$\mathbb{E}_{q(z|x_n)}[\log p(x_n|Z)] - D_{\mathrm{KL}}(q(Z|x_n)\|p(Z)) \quad .$$

As we assume the distribution $p(Z)$ being Gaussian with mean 0 and standard deviation 1, we can calculate the first term $\mathbb{E}_{q(z|x_n)}[\log p(x_n|Z)]$ by sampling multiple Z from $p(Z)$. Further, with this assumption we can easily calculate the D_{KL} term (see Question 1.5).

Question 1.8

The left-hand side of equation 11 is given as

$$\log p(x_n) - D_{\mathrm{KL}}(q(Z|x_n)||p(Z)).$$

We saw earlier that this term is a lower bound of the log-probability $\log p(x_n)$. When maximizing it in the training process, two desirable things happen:

1. $D_{\mathrm{KL}}(q(Z|x_n)||p(Z))$ gets lower.
   This means that $q(Z|x_n)$ gets more similar to $p(Z|x_n)$, i.e. the output of the encoder networks resembles more the prior distribution over Z. Consequently, the decoder network is trained from samples similar to those sampled from $p(Z)$.

2. $\log p(x_n)$ gets higher.
   This means that we optimize the log-probability of our data, i.e. we make it more likely that our Variational Autoencoder produces samples that resemble those of our dataset.

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

Question 1.9

The first loss $\mathcal{L}_n^{\mathrm{recon}}$ measures the negative log likelihood of a data sample x_n after it was transformed by the encoder network into a latent representation Z and then reconstructed by the decoder network. This can be regarded a *reconstruction* loss, because it is lower the better the output of the decoder resembles the input x_n.

The second loss term $\mathcal{L}_n^{\mathrm{reg}}$ is the Kullback-Leibler divergence between the encoder's output distribution $q_\psi(z|x_n)$ and the prior distribution over $P_\theta(Z)$. It can be regarded a *regularization* term, because it restricts the latent variables generated by the encoder to have certain properties (i.e. to look like vectors sampled from $P_\theta(Z)$). If this term would not be included in the loss function, the encoder would be likely to produce very different latent vectors for different images to make it easier for the decoder to distinguish them. However, as explained above, that would lead to the decoder not learning a dense mapping from latent space to image space which again would cause the decoder not to be generative.

Question 1.10

The total objective is given as

$$\mathcal{L} = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}_n^{\mathrm{recon}} + \mathcal{L}_n^{\mathrm{reg}}$$

with

$$\begin{aligned}
\mathcal{L}_n^{\text{recon}} &= \log p_\theta(x_n|Z) \\
&= \log \prod_{m=1}^{M} \text{Bern}(x_n^{(m)}|f_\theta(z)_m) \\
&= \log \prod_{m=1}^{M} f_\theta(z)_m^{x_n^{(m)}} (1 - f_\theta(z)_m)^{(1-x_n^{(m)})} \\
&= \sum_{m=1}^{M} \log(f_\theta(z)_m^{x_n^{(m)}}) + \log((1 - f_\theta(z)_m)^{(1-x_n^{(m)})}) \\
&= \sum_{m=1}^{M} x_n^{(m)} \log(f_\theta(z)_m) + (1 - x_n^{(m)}) \log(1 - f_\theta(z)_m)
\end{aligned}$$

(*Note that we can remove the expectation expression here because we are looking at just one sample.*)

and

$$\begin{aligned}
\mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}\left(q_\phi\left(Z|x_n\right)||p_\theta(Z)\right) \\
&= D_{\text{KL}}\left(\mathcal{N}(z_n|\mu_\phi(x_n), \text{diag}(\Sigma_\phi(x_n)))||\mathcal{N}(0, I_D)\right) \\
&= \frac{1}{2}\Big(\text{tr}(\text{diag}(\Sigma_\phi(x_n)) - I_D) + \mu_\phi(x_n)^T \mu_\phi(x_n) - log\big(|\,\text{diag}(\Sigma_\phi(x_n))|\big) \Big) \\
&= \frac{1}{2}\Big(\sum_{d=1}^{D} \Sigma_\phi(x_n)_d - D + \mu_\phi(x_n)^T \mu_\phi(x_n) - log\big(\prod_{d=1}^{D} \Sigma_\phi(x_n)_d\big) \Big)
\end{aligned}$$

1.5 The Reparametrization Trick

Question 1.11

(a) The gradient $\nabla_\phi \mathcal{L}$ is the gradient of the loss w.r.t the parameters of the encoder network. This particular gradient is necessary for updating the encoder parameters ψ when using stochastic gradient descent optimization.

(b) When performing stochastic gradient descent optimization on the model parameters, we need to back-propagate the computed error through the decoder and encoder network. However, the sampling operation on the encoder output is a non-continuous operation and therefore not differentiable. As the gradient for this "sampling layer" cannot be computed it cannot be backpropagated and hence, the parameters of the encoder network cannot be updated.

(c) To enable the computation of $\nabla_\phi \mathcal{L}$ we make use of the reparameterization trick, which is moving the sampling operation to an additional input layer. Concretely, instead of directly sampling

$$z_n \sim \mathcal{N}(\mu_\phi(x_n), \text{diag}(\Sigma_\phi(x_n))) \ ,$$

we add a sampled input $\epsilon \sim \mathcal{N}(0, I_D)$ and then compute z as

$$z_n = \mu_\phi(x_n) + \Sigma_\phi(x_n)^{\frac{1}{2}} \odot \epsilon \ ,$$

where the operator $\odot$ describes an element-wise product. Note that this form of sampling does not change the sampled z_n too much besides lowering their variance a little. However, it causes that given a fixed x_n and ϵ, z_n is computed with a deterministic function. Therefore, it has a derivative $\nabla_\phi \mathcal{L}$ w.r.t. the parameters of the encoder, which makes backpropagation possible and allows us to perform stochastic gradient descent optimization.

1.6 Putting things together: Building a VAE

Question 1.12

The Variational Autoencoder was implemented with the architecture given in the appendix of Kingma and Welling [2013]. The encoder network is a MLP with one linear hidden layer of size 500 and one

mean and one standard deviation linear layer of size D building upon it. Tanh activation is applied to the hidden layer. From the mean and standard deviation vector produced for an input image, we sample a latent vector z using the reparametrization trick given in the previous question. The decoder network is a MLP with two layers, of size 500 and $784 = 28 \cdot 28$ respectively. Tanh activation is applied to the hidden layer and sigmoid activation to the output layer.The loss function of the network is the one obtained in Question 1.10 . Note that just one sample is used to approximate the expectation term. The implementation can be found in `a3_vae_template.py`.

Question 1.13

Using at 20-dimensional latent space, we obtain the plot of the estimated lower-bounds (ELBO) of training and validation set that is given in figure 1.

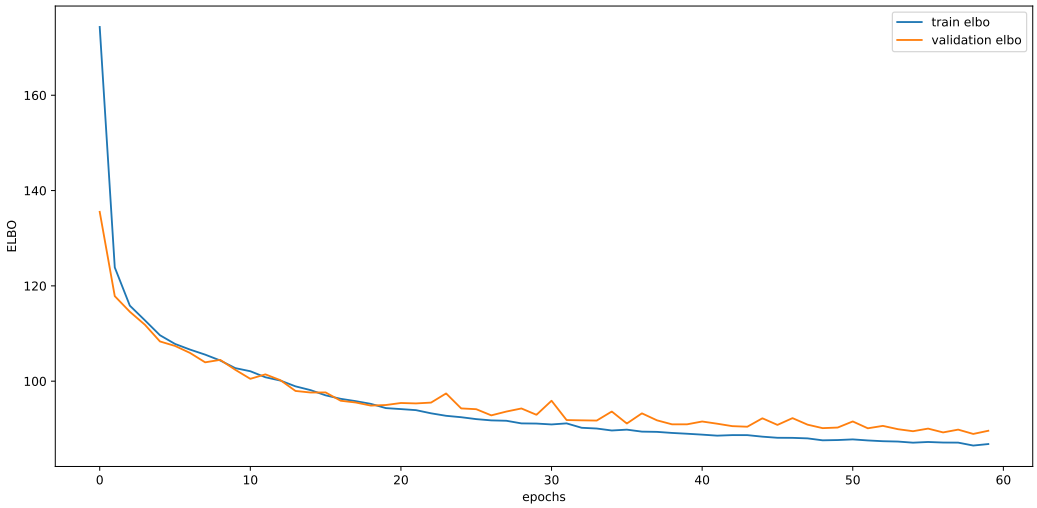

Figure 1: ELBO of training and validation set over 60 epochs of training.

Question 1.14

Figure 2 shows samples from the model trained with a two-dimensional latent space at different training epochs and before training. An improvement is clearly visible in term of non-blurriness and variety of generated numbers. However, the produced images after training remain slightly blurry. Note that the results for the model trained with a bigger latent dimension of 20 resulted in a smaller ELBO, but had a quite fragmentary appearance (see figure 3).

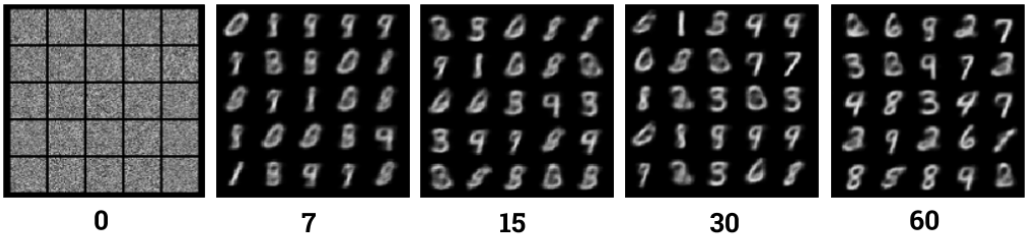

Figure 2: Images sampled with the VAE decoder network with 2 latent dimensions at different training epochs.

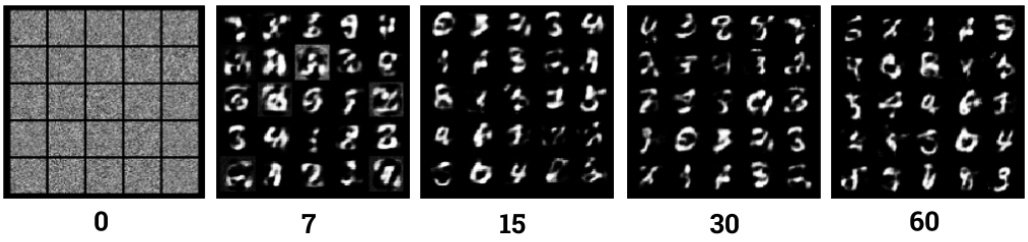

Figure 3: Images sampled with the VAE decoder network with 20 latent dimensions at different training epochs.

Question 1.15

The data manifold of the trained VAE with two-dimensional latent representation is depicted in figure 4.

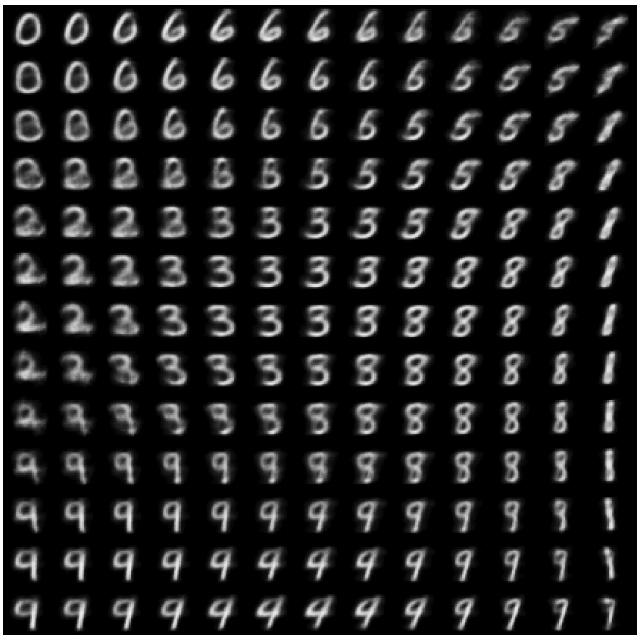

Figure 4: Data manifold of a Variational Autoencoder trained with a two-dimensional latent representation.

2 Generative Adversarial Networks

This part of the assignment is about Generative Adverserial Networks by Goodfellow et al. [2014].Given answers are based on information from the original paper.

Question 2.1

The generator network G receives as input a random variable z, which is sampled from the prior distribution $p_z(z)$. It outputs a vector of the same size as the ground data points of the training data set, i.e. a newly generated data sample.

The discriminator network D receives as input a data sample. This sample can either be the output of the generator network or a ground true one from the training data set. The network's output is a scalar $\in [0, 1]$ which indicates the probability of the input being a real data sample.

2.1 Training objective: A Minimax Game

Question 2.2

The GAN training objective is given as

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{p_{\text{data}}(x)}[\log D(X)] + \mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))] \ .$$

The first term of this objective is the expected log output of the discriminator network D for a ground true data sample X. This is a measure for how confidently the discriminator network identifies real data samples. D is trained to maximize this term, i.e. to confidently recognize original data samples as such and therefore output $D(X) = 1$ every time it receives a real data sample X as input.

The second term is the expected log of $1 - D(G(z))$, where $D(G(z))$ is the decoder output for a data sample produced by the generator network. This term is maximized if $D(G(z))$ is minimized, i.e. if the decoder network correctly identifies a generated sample as such and therefore outputs $D(G(z)) = 0$. This is what the decoder network is optimized for. In contrast, the term is minimized if the decoder mistakes generated samples for real ones, i.e. if $D(G(z)) = 1$, which is the objective of the generator network during training. To minimize the term, i.e. to achieve $D(G(z)) = 1$, the generator network has to produce output samples similar to the original data samples.

Question 2.3

As demonstrated in chapter 4 of Goodfellow et al. [2014], the minimax game between generator and discriminator network converges to its global optimum where the distribution of data samples produced by the generator network p_g is the same as the ground true data distribution p_{data}. At this point, the generator's output would be indistinguishable from real data samples. It is further shown in Goodfellow et al. [2014], that the optimal discriminator for a fixed generator network is given as

$$D(G(X)) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)} \ .$$

When we assume the global optimum of the minimax game ($p_g = p_{data}$) we obtain

$$D(G(X)) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_{data}(x)} = \frac{1}{2} \ ,$$

which shows that the discriminator network outputs 0.5 for every input sample (either generated or original) at the global optimum of the minimax game. Inserting this into the value function $V(G, D)$, we obtain:

$$\begin{aligned} V(D,G) &= \mathbb{E}_{p_{\text{data}}(x)}[\log D(X)] + \mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))] \\ &= \mathbb{E}_{p_{\text{data}}(x)}[\log \frac{1}{2}] + \mathbb{E}_{p_z(z)}[\log(1 - \frac{1}{2}))] \\ &= \log(\frac{1}{2}) + \log(\frac{1}{2}) = -2\log(2) = -\log(4) \end{aligned}$$

Question 2.4

Early on during training, the generator network produces data samples of very poor quality. Therefore, the discriminator can quite easily learn to distinguish between true and generated data samples and is likely to reject all generated samples quite fast. Consequently, the term $\log(1 - D(G(Z))$ is close to 0, the backpropagated gradient vanishes and G cannot improve. To solve this problem, we can train G for maximizing $\log D(G(z))$ instead of minimizing $\log(1 - D(G(Z))$. Note that this changes nothing about the overall training objective of the model. However, this way, when the discriminator rejects all generator samples, the gradient of G's loss function will be large and G can improve. Moreover, if there are only small changes in the output of D for a generated sample, this will result in rather big changes in the gradient of G's loss. This means, G will get a positive feedback for even small positive changes in its outputs.

2.2 Building a GAN

Question 2.5

For the implementation of encoder and discriminator network I used the model architectures suggested in the provided code template. The generator network has a tanh activation on the output layer, such that the generated images have the same value range as the ground true ones. The discriminator network has sigmoid activation on its output, such that the produced scalar is in $[0, 1]$ and can be interpreted as a probability.

To prevent the discriminator network from getting too good too fast, I forwarded only one batch of data through the network per time step. This batch consisted of half original data samples and half generated samples. To further improve the quality of the generated images, I make use of smooth labels as suggested in Salimans et al. [2016]. As another improvement, z is sampled from a Gaussian distribution instead of from the Uniform. This was suggested in White [2016].

The implementation can be found in `a3_gan_template.py`. The loss curves of both networks over 200 training epochs are displayed in figure 5.

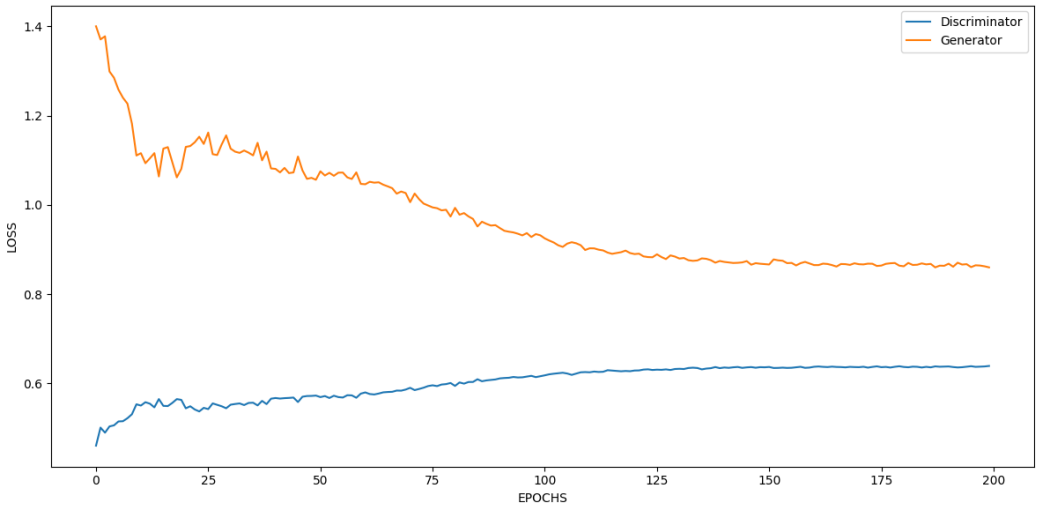

Figure 5: Loss of the generator and the discriminator network measured over 200 training epochs on training data.

Question 2.6

Figure 6 displays images sampled from the network at different epochs of the training. We can see that the initial images are just random noise. However, the generator network quickly learns the produce rather blurry numbers (epoch 25). Over the further steps, the generated images get sharper and less noisy.

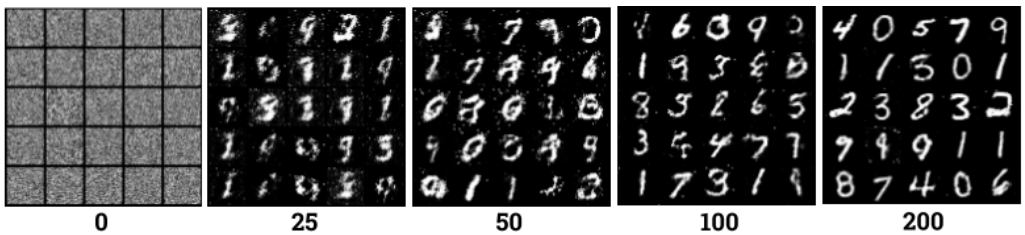

Figure 6: Images sampled from the generator network at different training epochs.

Question 2.7

The result of the 7-step interpolation in latent space between two digits is given in figure 7.

Figure 7: Interpolation in latent space between two digits.

3 Generative Normalizing Flows

This chapter is about flow-based generative models as described by Rezende and Mohamed [2015]. The answers are based on the original paper as well as on the paper on RealNVP by Dinh et al. [2016] and the blogpost by Lilian Weng, linked in the task description.

3.1 Change of Variables for Neural Networks

Question 3.1

For the case that $x \in \mathbb{R}^m$ (i.e. is a multivariate random variable) and f is an invertible mapping $f : \mathbb{R}^m \longrightarrow \mathbb{R}^m$ we have:

$$z = f(x) \qquad x = f^{-1}(z) \qquad z \sim p(z)$$

$$p(x) = p(z) \left| \det \frac{dz}{dx} \right| = p\left(f(x)\right) \left| \det \frac{df}{dx} \right|$$

$$\log p(x) = \log p(z) + \sum_{l=1}^{L} \log \left| \det \frac{dh_l}{dh_{l-1}} \right|$$

Question 3.2

Since the determinant is only defined for quadratic functions and we know that the overall $f : \mathbb{R}^m \longrightarrow \mathbb{R}^m$, we can conclude that for all $h_k \quad \forall k \in \{1, .., l\}$ with

$$z = f(x) = h_l \circ h_{l-1} \circ \ldots \circ h_1(x)$$

it has to hold that $h_k : \mathbb{R}^m \longrightarrow \mathbb{R}^m \quad \forall k \in \{1, .., l\}$. Further, every h_k has to be an invertible and smooth function in order to make to overall flow function f invertible and smooth.

Question 3.3

There are two computational issues that arise when optimizing the network for the previously defined objective:

1. Calculating the determinant for a $N \times N$ Jacobian matrix is typically of runtime complexity $O(N^3)$. This is very expensive, especially when we think about that we have to compute the determinant once for every h_k with $k \in \{1, ..., l\}$ in every training step. Additionally, in this assignment we have a quite high $N = m = 784$.
2. Calculating the function inverse for a function approximator, which is a weight matrix of size $N \times N$, also has a computational complexity of $O(N^3)$ when done with the Gauss-Jordan elimination algorithm.

Question 3.4

When we fit the continuous flow-based model to discrete data, such as image data with pixel values in $\{0, ..., 255\}$, we will end up with a spiky distribution that puts all probability values > 0 on discrete data points. One solution for this problem is to initially transform the discrete data distribution into

a continuous distribution. There are several methods to do this, however, the method we use for the implementation is the *Uniform Dequantization* described by Ho et al. [2019]. With this method, uniform noise is added to each data point over all dimensions. Concretely, we replace each data point $x \in \mathbb{R}^m$ with

$$x \leftarrow x + u \quad \text{where} \quad u \sim \mathcal{N}(0, I_m) \ .$$

As a consequence, we learn an approximation of the data distribution that has no spikes on discrete data samples but is continuous because it is trained with the non-discrete data samples surrounding the ground true discrete ones.

3.2 Building a flow-based model

Question 3.5

As given in the implementation template, at training time we input a real data sample $x \in \mathbb{R}^m$ to the flow-based model. This data sample is passed through the flow and finally all intermediate Jacobian log-determinants are combined with the probability of the latent representation $p(z)$ to the output of the network: the log-probability $log(p(x)) \in \mathbb{R}$. The negated log-probability $-log(p(x))$ represents the network's loss and is subsequently backpropagated through the network to update its layers.
At inference time, the network receives as input a latent vector $z \in \mathbb{R}^m$ sampled from the assumed latent distribution (usually Gaussian). This vector is passed through the reverse flow and we obtain as output a new data point $\hat{x} \in \mathbb{R}^m$.

Question 3.6

Training a flow-based model proceeds as follows:

1. A data sample x is selected as input data.
2. The sample is sequentially passed through all layers (i.e. functions h_k for $k \in \{1, .., l\}$) until we receive the latent representation z as
$$z = h_l \circ h_{l-1} \circ \ldots \circ h_1(x) \ .$$
3. In parallel to step 2, we compute for each layer h_k the corresponding $\log \left| \det \frac{dh_l}{dh_{l-1}} \right|$.
4. After having passed x through all flows, we compute the log probability of the obtained latent vector $z \in \mathbb{R}^m$ as
$$\log(p(z)) = log(\mathcal{N}(z|0, I_m))$$
because we assume the latent distribution p_z to be a Gaussian.
5. We can then combine our previous results to compute the flow-network's loss-function as the negative log-probability of x with
$$-log(p(x)) = -\left(\log p(z) + \sum_{l=1}^{L} \log \left| \det \frac{dh_l}{dh_{l-1}} \right| \right) \ .$$
6. The loss is backpropagated through the flow network and the functions/layers h_k are updated accordingly.

As usual, this process is repeated for multiple training data samples over several epochs. Note that in an actual implementation of this algorithm x is not a single data sample but rather a batch of samples.

At inference time, we generate a new data sample with the following steps:

1. We sample a new latent variable z from the latent distribution $\mathcal{N}(0, I_m)$.
2. The sampled z is passed through the inverted flow. For this purpose, every h_k needs to be inverted. We obtain the generated data sample $\hat{x}$ as
$$\hat{x} = h_1^{-1} \circ \ldots \circ h_{l-1}^{-1} \circ h_l^{-1}(z) \ .$$

Question 3.7

We implement the Real-valued Non-Volume Preserving (RealNVP) model by Dinh et al. [2016]. This version of a flow-based generative network models the bijections h_k as so-called *coupling-layers*. The input of such a layer is split into two parts (in our case of equal size). The first part stays the same, the second part is transformed with a scale-and-shift transformation. The advantage of using a coupling layer as intermediate function h_k is twofold: First, it is very easy to invert the transformation operation, which means we have no almost no additional computational cost when sampling a new $\hat{x}$. Second, the Jacobian determinant corresponding to a coupling layer is very easily calculated. This is because the Jacobian is a lower triangular matrix and therefore its determinant can be computed as just the product of all values on the diagonal. To decide which coupling layer inputs are kept as they are and which are transformed, we make use of checkerboard-masks. Specifically, we alternately use reversed checkerboard-masks and an equal number of coupling layers. As already mentioned above, we make use of uniform dequantization to obtain an approximated data distribution without spikes. Before an dequantized image x is fed to the model, it's values are normalized from the interval $[0, 255]$ to $[0, 1]$. Consequently, we have to transform the values from a sampled $\hat{x}$ back to $[0, 255]$. The implementation can be found in `a3_nf_template.py`.

Question 3.8

When training the flow-based model, I receive the loss-curve shown in figure 8. As expected, the model reaches below 1.85 bits per dimension after 40 epochs of training. Figure 9 further shows images generated with the model after on epoch of training and after the training is completed. We can see a clear improvement over the time of training. However, the final generated images remain to a small extent fragmentary.

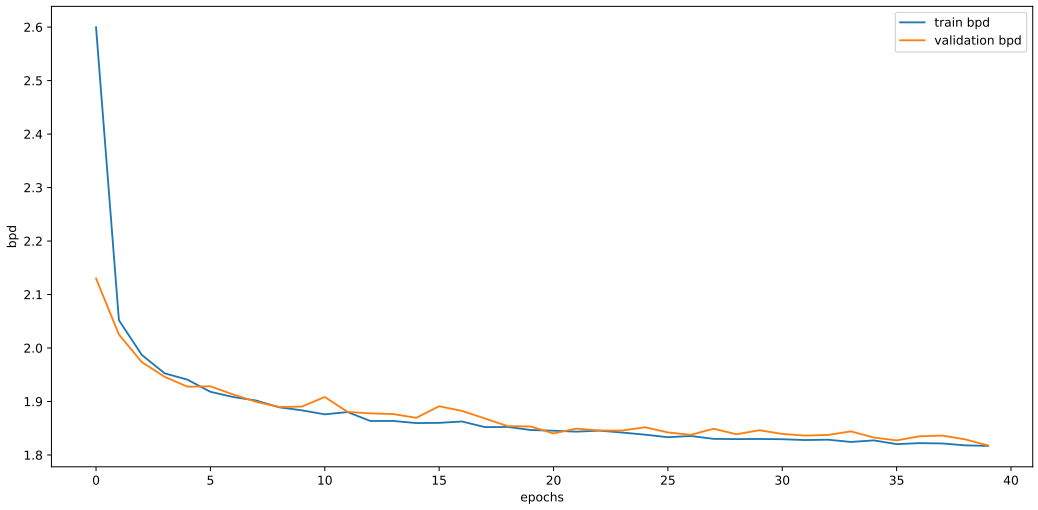

Figure 8: Loss of the flow-based RealNVP model measured in bits per dimension (bpd) over 40 training epochs on training and validation data.

4 Conclusion

In this assignment, three methods for generative modelling were extensively described and explored. The first model, the Variational Autoencoder, maximizes the data log-likelihood by maximizing the evidence lower bound (ELBO) and uses the learned approximate inference to generate new data samples. The images resulting from my implementation were unfortunately a bit blurry, but definitely had a natural appearance. However, the blurriness is a known problem for VAEs (see Goodfellow et al. [2016], chapter 20). In contrast, Generative Adversarial Networks learn to create data samples using a minimax game between a generator and a discriminator network. The resulting images are clearly the best result in the scope of this report. They appear sharp-edged and natural. It is to note

(a) After one epoch of training (b) After training is completed

Figure 9: Images sampled from the flow-based RealNVP model.

that the GAN was harder to train than the other two models, because its training process is much more instable. To train it successfully, some adjustments had to be made, which required some further reading. Finally, flow-based models are a more recent development in generative modelling. They consist of a chain of invertible functions and learn the data distribution by simply optimizing the negative log-likelihood. The results that I achieved in this report using the Real NVP model by Dinh et al. [2016] were unfortunately rather fragmentary, but they still clearly resembled the input data.

References

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

Stefano Ermon. CS 228 - Probabilistic Graphical Models, Winter 2018-19. URL `https://ermongroup.github.io/cs228-notes/inference/sampling/`.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`.

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242, 2016.

Tom White. Sampling generative networks. *arXiv preprint arXiv:1609.04468*, 2016.

Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*, 2015.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real NVP. *arXiv preprint arXiv:1605.08803*, 2016.

Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. *arXiv preprint arXiv:1902.00275*, 2019.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.