Deep Learning Assignment 3, Fall 2020

Blazej Manczak
University of Amsterdam
blazej.manczak@student.uva.nl

1 Variational Auto Encoders

1.1 Latent Variable Models (food for thought)

The VAE and standard autoencoders sound similar but are quite different. Autoencoders learn the compressed data representation with an objective of minimizng the reconstruction error. They are mostly used for pre-training, projecting data into lower dimension for plotting or simply compression. Variational autoencoders addtitionally assume and try to estimate the underlying distribution that generates the data. Since we estimate the parameters of the distribution, we can sample new data from it (i.e. the model is generative, unlike regular autoencoder)! VAE's encapsulate the capabilities of the regular autoencoder and can be used instead of one for the purposes mentioned above.

1.2 Decoder: The Generative Part of the VAE

Question 1.1

To sample from $p(x_n|z_n)$ we use forward sampling. In this case that amounts to first sampling the latent z_n from $p(z_n)$. Then we get the value of $f_{\theta(z_n)}$ by running our z_n through the decoder. Since $f_\theta(z_n)$ contains the parameters for our Bernoulli distribution, we can simply evaluate $Bern(x_n^{(m)}|f_{\theta(z_n)_m})$ for each pixel m. Once we have this we perform the multiplication $\Pi_{m=1}^{M}$ and we arrive at $p(x_n|z_n)$.

Food for thought: Even though the assumption about $p(Z)$ is simplistic, it is not such a restrictive assumption. It is not because any distribution in D dimensions can be approximated by a taking a set of D normally distributed variables and mapping them through sufficiently intricate function such as a neural network.

Question 1.2

Monte-Carlo Integration is insufficient because the probability $p(x_n|z_n)$ will be very low for most of the z's sampled from $p(z_n)$, as can be seen in Figure 2 in the assignment pdf. This random sampling would result in a poor estimate of the expectation which in turn would make efficient training impossible. Only a part of the z space will be relevant for a given x_n. The problem gets more severe as the dimensionality of z increases because is is less and less likely for a z sampled from $p(z)$ to be relevant for x_n.

Question 1.3

Given equation (11) from the pdf assignment, we see that two the same distributions will result in 0 KL divergence, for example setting $(\mu_q, \mu_p, \sigma_q^2, \sigma_p^2)$ to $(0, 0, 1, 1)$. On the other hand the KL divergence will be large for wildly different distributions, for example when $(1000, 0, 1000, 1)$

Question 1.4

The right hand side (RHS) of Eq. 14 is a lower bound because the expression $KL(q(Z|x_n)||p(Z|x_n))$ on the left-hand side (LHS) is always positive. Thus when we eliminate a subtraction of a positive value on the LHS we get that LHS $\geq$ RHS. We must optimize the lower-bound instead of $logp(x_n)$ because the direct optimization of $logp(x_n)$ is intractable. We make it tractable by swapping $log\mathbb{E}$ to $\mathbb{E}log$. Now we can simply take the expectation of a logarithm of a function, which is especially convenient when the function is from the exponential family.

Question 1.5:

When the lower bound (RHS of eq. 14) is pushed up two things can happen on the RHS:

1. The difference between the true posterior and the approximate posterior decreases, i.e. $KL(q(Z|x_n)||p(Z|x_n))$ gets smaller.
2. the log probability of the data point increases, i.e. $logp(x_n)$ increases

Both of these things are good for our optimization process!

Question 1.6:Reconstruction and Regularization loss names

The name *reconstruction* loss makes sense because optimizing this term encourages reconstructed images that maximize the log likelihood of the data (produce output similar to the training data).

The name *regularization* loss makes sense because optimizing this term ensures that our approximate distribution is not too different from he prior distribution.

Question 1.7: Reconstruction and Regularization loss formulas

For the **reconstruction loss** we have:

$$\mathcal{L}_n^{recon} = -\mathbb{E}_{q_\phi(z|x_n)}[logp_\theta(x_n)|Z)] \approx -\frac{1}{K}\sum_{k=1}^{K} logp_\theta(x_n|Z_K)$$

Here I wrote a sum over k but one usually approximates the expectation with only one example. For **regularization loss** we simply evaluate the KL divergence between two Gaussian distributions (adaptation of eq 9):

$$\mathcal{L}_n^{reg} = D_{KL}(q_\phi(Z|x_n)||p_\theta(Z)) = \frac{1}{2}(log\frac{|\Sigma_p|}{|\Sigma_q|} - d + tr[\Sigma_p^{-1}\Sigma_q] + (\mu_p - \mu_q)^T\Sigma_p(\mu_p - \mu_q))$$

When one Gaussian is a (multivariate) unit Gaussian this comes down to:

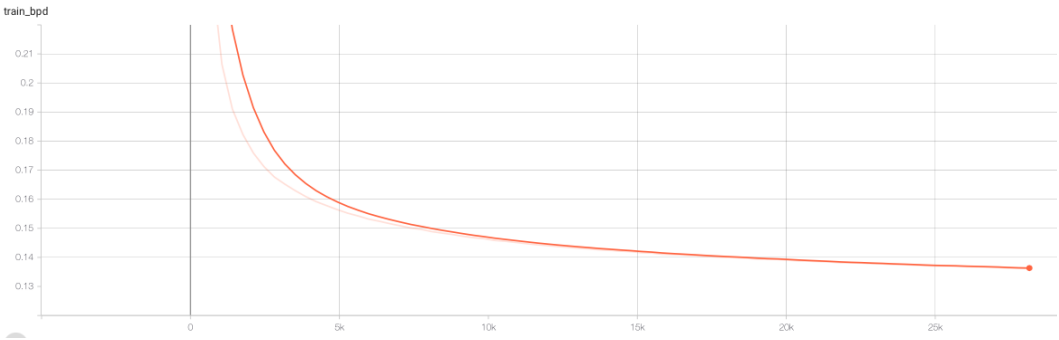
$$= \frac{1}{2}(-log|\Sigma_q| - d + tr[\Sigma_q] + \mu_q^T\mu_q)$$

Question 1.8: Reparemetrization trick

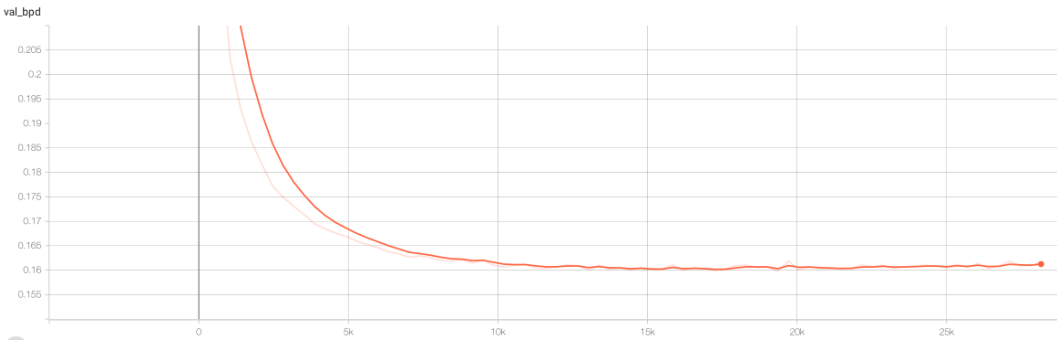
The act of sampling prevents us from calculating $\nabla_\phi\mathcal{L}$ because the sampling operation itself, which is not differentiable, depends on parameter ϕ wrt to which we would like to take the derivative. Hence, we can not back-propagate. This issue is neatly resolved by the reparameterization trick which essentially moves the source of stochasticity to an external random variable that does not depend on ϕ. Instead of sampling z directly from $\mathcal{N}(z_n|\mu_\phi, \Sigma_\phi)$ we instead construct our variable z to be $\mu_\phi + \Sigma_\phi \cdot \epsilon$, where $\epsilon \sim \mathcal{N}(0, I)$. This way the randomness is not associated with the encoder (parameters) anymore and the gradients can freely flow through $\mu_\phi + \Sigma_\phi$!

Question 1.9: VAE description

Given all that, we can build a variational autoencoder. I've trained it on the binarized MNIST dataset, using an MLP for the encoder and decoder. I've used the default settings besides the hidden sizes:



(a) Train BPD



(b) Validation BPD

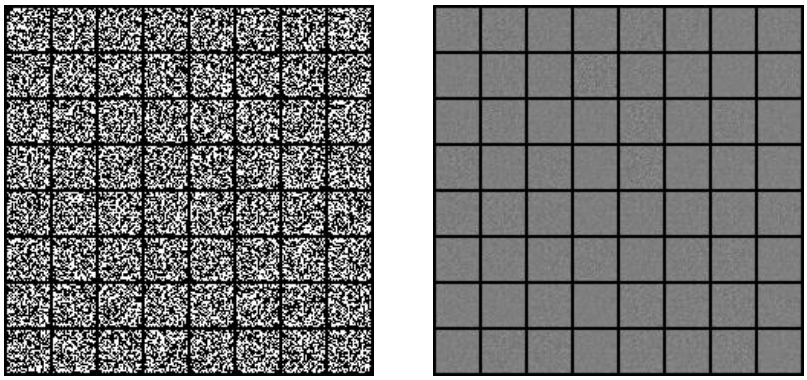
Figure 1: Training and Validation BPD

both the encoder and the decoder have two hidden layers with sizes 512,256 , the latent dimension had size 20, batch size 128, learning rate $1e-3$. The network was trained for 80 epochs. In Figure 1 please find the plots of training and validation BPD:

We see that training BPD converges around 0.13-0.14, and validation BPD around 0.16. It is not visible on the plot because of big scale difference but both start around 1.

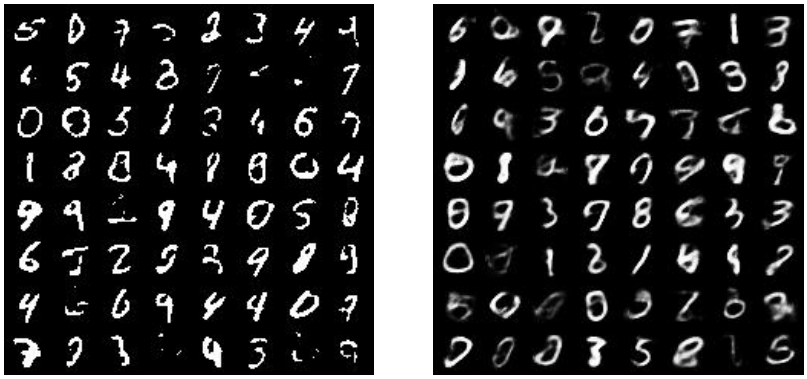
Question 1.10: Sample generations

Below please find the sample generations before training, after 10 and 80 epochs. We see an improvement in quality of samples. Naturally before training we get noise, as seen on Figure 2. After 10 epochs (Figure 3) the generations are mostly legible numbers but they sometimes miss a part of number. After 80 epochs (Figure 4) this issue is mostly eliminated and the sample look good.



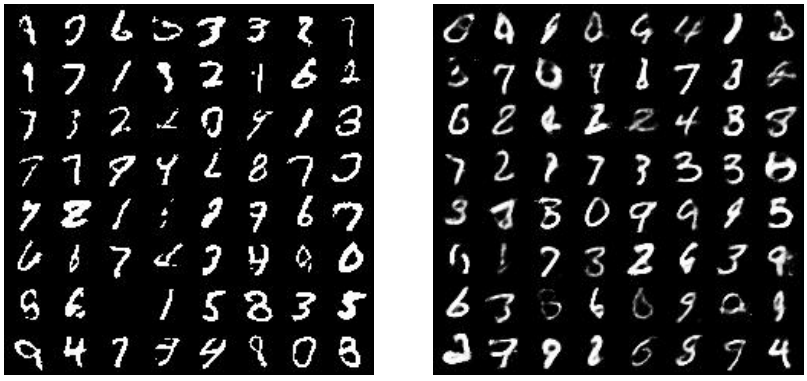
(a) Sample image (b) Mean image

Figure 2: Sample and mean image before training (MLP)



(a) Sample image (b) Mean image

Figure 3: Sample and mean image after 10 epoch (MLP)



(a) Sample image (b) Mean image

Figure 4: Sample and mean image after 80 epoch (MLP)

Question 1.11: Visualizing the data manifold

In Figure 5 we can observe the data manifold from a 2 dimensional latent space. We see that the digits smoothly transition one into another, like for example 3 into an 8 or a 2.

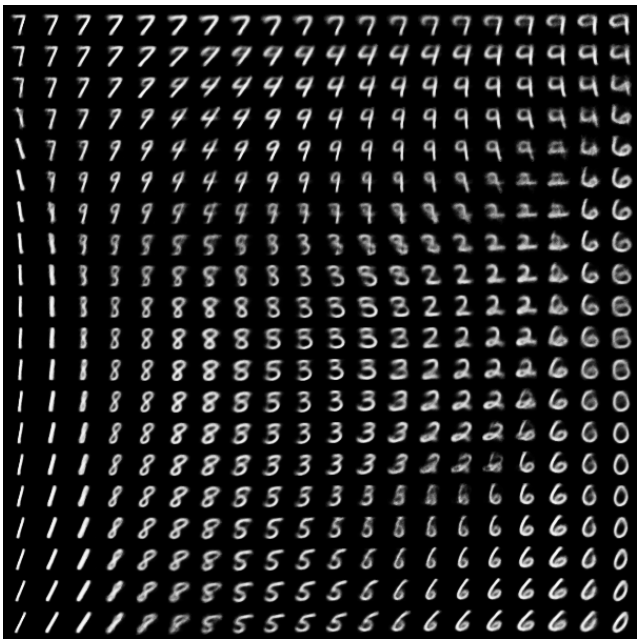
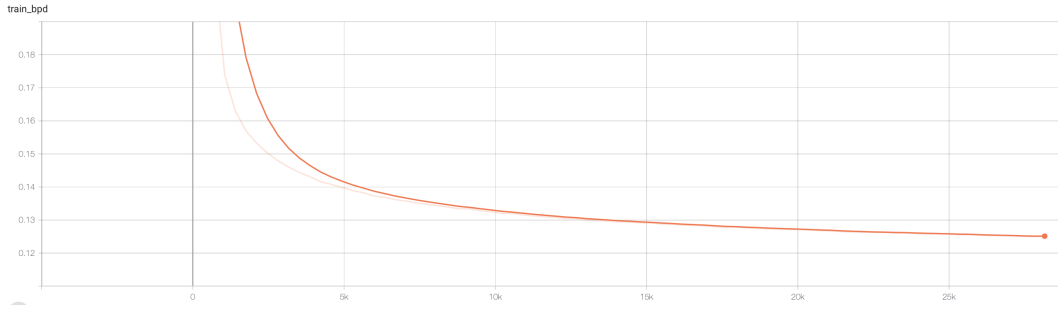


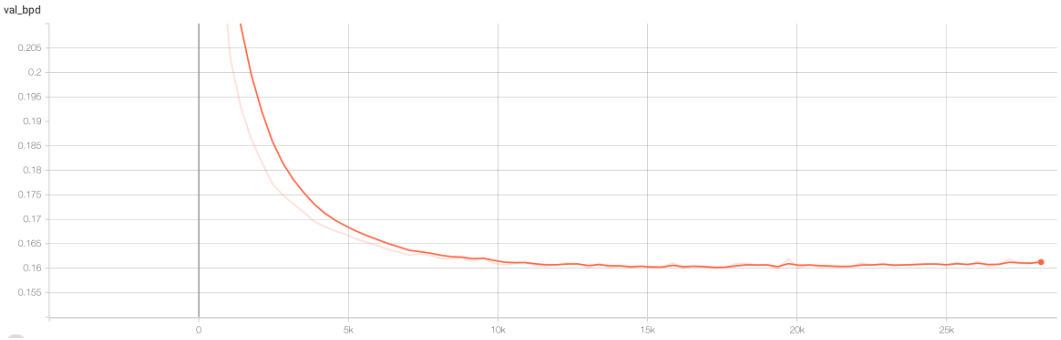
Figure 5: Data manifold from 2-dimensional latent space

Question 1.12 (Bonus): CNN as encoder/decoder

I've also implemented the CNN variant for encoder and decoder. I've used the same architecture as in Tutorial 9. In Figure 6 I plotted the BPD score for validation and training. We again see that the validation BPD converges to around 0.16 and training BPD between 0.13 and 0.14, similarly the MLP variant. It is noteworthy that this architecture uses over 4 times less parameters than the one I used for MLP.



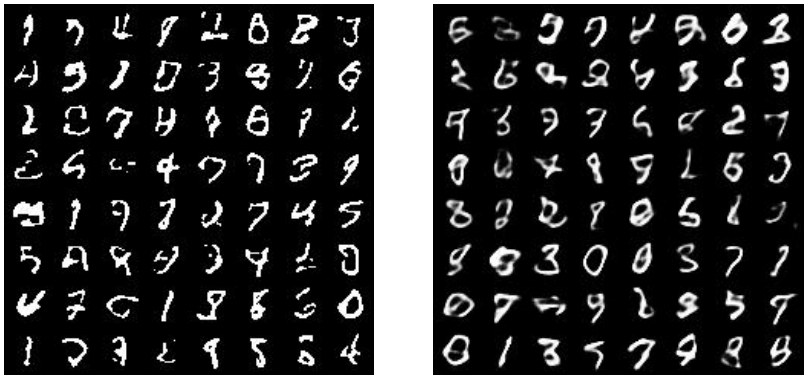
(a) CNN Train BPD



(b) CNN Validation BPD

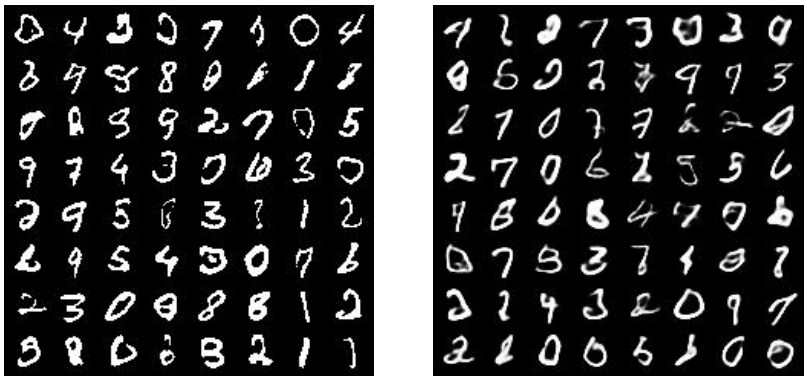
Figure 6: Training and Validation BPD of the CNN architecture

In Figures 7 and 8 I've plotted the sample and mean images before after 10 and after 80 epochs of training respectively.



(a) Sample image (b) Mean image

Figure 7: Sample and mean image after 10 epoch (CNN)



(a) Sample image (b) Mean image

Figure 8: Sample and mean image after 80 epoch (CNN)

Again, after 10 epochs we see that many numbers have strange artifacts but are mostly legible. Subjectively the generations at this stage look better than for MLP. After 80 epochs we see that generations have improved and almost all numbers are legible. However the "font" seems peculiar and different than for MLP. That is probably due to the nature weight sharing nature of convolutional layers. All in all, the CNN implementation of the VAE is more efficient with the parameters and yields generations of similar quality.

2 Generative Adverserial Networks

Question 2.1: GAN training objective

a) In the GAN training training objective $\mathbb{E}_{p_{data}(x)}[logD(X)]$ refers to maximizing discriminator's power (log-likelihood) to recognize that the data sampled from the real dataset should be classified as a real (not generated) image. This part refers solely to the discriminator.

On the other hand $\mathbb{E}_{p_z(z)}[log(1-D(G(Z))]$ refers to both generator and the discriminator. In the min step the generator is encouraged to produce samples that confuse the discriminator, i.e. are similiar to the true data. Then in the max step the discriminator is optimized to distinguish the artificially generated samples.

b) When the training converges, $p_{data} \approx p_{generative}$ and $V(D,G)$ becomes

$$\mathbb{E}_{x\sim p_{data}} log \frac{p_{data(x)}}{p_{data}(x)+p_{generative}(x)} + \mathbb{E}_{x\sim p_z} log \frac{p_{generative(x)}}{p_{data}(x)+p_{generative}(x)} = -2log2$$

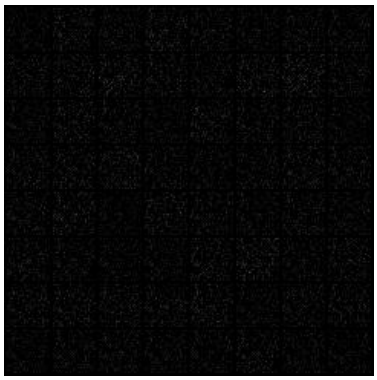
This makes sense, because at optimally the generator will converge on the true distribution and the best the discriminator could do would be guessing.

Question 2.2: No gradients in early steps

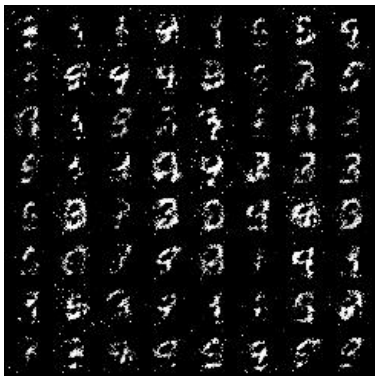
The term $log(1-D(G(Z)))$ might pose problems because early on the generator is terrible and generates samples that are very different from true samples, making the job for the discriminator easy. In this case $log(1-D(G(z))$ saturates and there aren't sufficient gradients for G to learn well. Instead we can train G to maximize $logD(G(Z))$ which will provide stronger gradients, especially in the beginning of the training.

Question 2.3: Implementing a GAN

I've trained the GAN with the default parameters of the assignment: the latent vector z has 32 dimensions, the generator has three hidden layers with $128, 256, 512$ hidden sizes respectively. The discriminator uses two hidden layers with dimensions $512, 256$ respectively. After each linear layer in the generator and the discriminator dropout layer is used with dropout rate 0.1 and 0.3 respectively.



(a) Before training epochs



(b) After 10 epochs

(c) After 250 epochs

Figure 9: Sample GAN images during different stages of training

Adam optimizer is used with learning rate $2e-4$ and batch size 128. GAN has been trained for 250 epochs. Please find the sample generations before training, after 10 epochs and after 250 epochs in Figure 9.

As expected, we do not get anything before the training starts. After 10 epochs the generations seem to be forming number like shapes with little dust. Finally, after 250 epochs we see that the quality of the generated images is high: the numbers are clearly distinguishable and there not that many artifacts. The diversity of the numbers also seems to be reasonable suggesting that the model did not suffer from (severe) mode collapse.

Question 2.4: Interpolating latent noise

In Figure 10 please find the sample interpolations in the latent space of the fully trained GAN model:

Figure 10: Linear interpolation in latent space between 4 images with 5 interpolation steps.

We see rather smooth transitions from one generated number (most left) to the second generated number(most right) with a different number showing up in the interpolation steps. For instance, in the last row we see that by interpolating the latent representation yielding a 9 and an 8 we get number 3. However we also see that some transitions are not as smooth, like for example in the second row.

Question 2.5: Problems in GAN training

Fortunately I have not encountered any significant problems in training the GAN. I will describe one of the most prevalent issues while training GAN's, namely mode collapse.

Mode collapse occurs when the generator maps multiple different z values to the same output. The most prevalent scenario is partial mode collapse which results in the generator producing outputs of small diversity. Specifically, instead of yielding the generations containing all the modes of the training set, the generator produces samples from one mode at the time. Once the discriminator learns to reject samples from that mode, the generator switches to a different mode. This process then can go in cycles, resulting in peak-and-valley loss curve for the generator. One possible way to address this is minibatch discrimination. This allows the discriminator to compare an example to the minibatch of real and generated samples. Then the discriminator can use the information on similarity to improve its detection of fake images, effectively penalizing mode collapse.

3 Generative Normalizing Flows

Question 3.1: Change of variable example

In our case $z \sim \mathcal{U}[a, b], z = f(x) = x^3$ Thus we have that

$$p_x(x) = p_z(z)|\frac{df(x)}{dx}| = \frac{1}{b-a}|3x^2|$$

for $x \in [\sqrt[3]{a}, \sqrt[3]{b}]$ and 0 otherwise

Integrating over x yields yields 1: $\int_{-\infty}^{\infty} \frac{1}{b-a}|3x^2|dx = \frac{1}{b-a}\int_{\sqrt[3]{a}}^{\sqrt[3]{b}} 3x^2 dx = \frac{1}{b-a}[x^3]_{\sqrt[3]{a}}^{\sqrt[3]{b}} = 1$

The new density has a parabolic shape.

Question 3.2: Limitations of Normalizing Flow models

a) In order to be able to compute the determinant of the Jacobian the transformation functions $h_l \; \forall l$ have to be invertible, continuous and have the same dimension. Otherwise we would to be able to compute the inverse, the jacobian and the determinant thereof.

b) Given those constraints one must make smart choices to make the optimization process smooth. Computing the determinant of the jacobain must be efficient so one should wisely choose the transformations. The inverse transformation should also be easily computable, especially if one is interested in efficient sampling. Moreover, if the flow is trained on high resolution data, sampling data points at test time might be computationally expensive. That is because the transformations must

preserve the dimensions. Even if our transformations are efficient, performing many high dimensional computations will take a toll on our processing unit.

Question 3.3: Building a flow-based model

a) Normalizing flow relies on transformations in a continuous space. When we apply normalizing flows to discrete data as a resulting density we get arbitrarily high likelihoods, called δ picks, around the discrete values. The learnt density does not provide us with a distribution over the discrete points as it sums up to arbitrarily high number, not to 1 like a valid probability density. A way to fix it is to apply a dequantization procedure. It adds (uniform) noise to each discrete value. Then the discrete value can be modeled by a distribution over an interval.

b) Before we start to train a flow based model we need to define our simple prior distribution and type of flow layers, i.e. the invertible functions f mentioned above. Popular choices include a unit Gaussian and coupling layers respectively. With that we can start training. For training and validation, we estimate the density in the forward direction by applying our flows to the input X in sequence, a pre-specified number of times. As an output of this step we get 2 ingredients needed for the density estimation: the probability of the transformed input(s) given our prior and the sum of changes in volume caused by the consecutive transformations (sum of the determinants of the jacobians). The estimated density $logp(x)$ is used to evaluate the model. To sample from the model we sample from our prior and apply the inverse transformations.

4 Conclusion

In this assignment we've investigated 3 different generative models: VAEs, GANs and Flow based models. Each model uses a latent representation $\mathbf{z}$ in a slightly different manner. In turn, the different approaches yield different benefits and downsides.

In VAEs we model the encoder $q(z|x)$ that maps input x to a distribution over the latent variable z. Then we sample z from this distribution and learn the decoding network that maps z to to the reconstruction of the input. VAEs model the lower bound of the data loglikelihood. They offer stable training, fast sampling and learn a compressed representation. However the generations are often inferior to the ones offered by GANs.

In GANs we train the discriminator and the generator in a minimax game fashion. The generator learns to generate realistic samples given a sample from a prior noise distribution $p(z)$. The discriminator then tries to distinguish between real and generated samples. GANs offer one of the best quality generations and fast sampling, however their training is unstable, there is no model data likelihood and they do not learn a compressed representation.

Finally, flow based models are constructed by applying a series of invertible transformations that sequentially mold the latent distribution z into the input x. They model the exact data likelihood, have stable training and have reasonably fast sampling however they do not have a compression mechanism.