Deep Learning practical 3

Jochem Soons
MSc Artificial Intelligence
University of Amsterdam
UvA-id: 11327030
Date: 23-12-2020
jochem.soons@student.uva.nl

1 Variational Auto Encoders

Question 1.1

The sampling process essentially contains three steps:

1. We sample z_n from our prior distribution: $z_n \sim \mathcal{N}(0, \boldsymbol{I}_D)$
2. We pass the latent representation z_n through our decoder network $f_\theta(z_n)$ to compute the mean value for every pixel $m \in M$ of our output image size
3. Using the computed means we can sample the pixel values $x_n^{(m)}$ from the Bernoulli distribution Bern($f_\theta(z_n)$) for every $m \in M$. Now we have sampled a new image from our decoder.

Question 1.2

Monte-Carlo integration using samples from $p(z_n)$ (random sampling) is not efficient because a lot of sampled z_n values will be in low density regions of our distribution $p(z|x)$. This can also be seen in figure 2 of the assignment PDF (only a small portion of the probability distribution $p(z)$ has a high density $p(z|x)$). So using this method leads to a lot of unimportant likelihood (near-zero) terms that will not contribute to approximating $\log p(x_n)$. Put differently, we need to sample from high density regions of $p(z|x)$ to get an accurate approximation but we are not guaranteed to sample from these regions if we use this random sampling approach. This in turn leads to an unreliable high-variance estimator, meaning that if we repeat the sample process multiple times we are likely to obtain very different results in approximating $\log p(x_n)$.

To get somewhat reliable and low-variance results, we would need to sample a lot of values. Moreover, the inefficiency of this approximation method will scale with the dimensionality of z: if z increases in dimensionality the ratio of high density regions $p(z|x)$ to regions where $p(z) > 0$ increases (so even a smaller portion of $p(z)$ contains high density regions which we need for approximating), which means that we would need even more samples to get an reliable approximation of $\log p(x_n)$. Put differently, a higher dimensionality of z would mean more meaningless sampling from low density regions so more samples are needed to make an approximation.

Question 1.3

The KL divergence $\mathrm{D}_{KL}(q||p)$ measures the similarity between the two distributions p and q (how far they are apart). So when both distributions are exactly the same (e.g. they are both standard normal distributions), the KL divergence would be 0. An example:

$\mathrm{D}_{KL}(q||p)$ = 0 for $(\mu_q, \mu_p, \sigma_q^2, \sigma_p^2)$ = (0, 0, 1, 1)

On the other hand, the KL divergence would be very large for two distributions that are very different in terms of mean and variance (e.g. p is is the standard normal distribution but q is a distribution with mean 1000 and variance 50). An example:

$D_{KL}(q||p)$ = 523 for $(\mu_q, \mu_p, \sigma_q^2, \sigma_p^2)$ = (1000, 0, 50, 1).

Question 1.4

We know that the KL divergence between two distributions can only be positive or zero, so this also holds for the KL divergence term on the left hand side of equation 14: $KL(q(Z|x_n)||p(Z|x_n)) \geq 0$. This in turn means that:

$\log p(x_n) - KL(q(Z|x_n)||p(Z|x_n)) = ELBO \rightarrow \log p(x_n) \geq ELBO.$

So the difference between the ELBO (right hand side of equation 14) and the log likelihood $\log p(x_n)$ is exactly $KL(q(Z|x_n)||p(Z|x_n))$, which is a positive term. So from that we can conclude that the ELBO is by definition a lower bound of our objective $\log p(x_n)$. Moreover, in the training process we must optimize this lower bound instead of optimizing the $\log p(x_n)$ directly because the expression of $\log p(x_n)$ contains intractable terms, i.e. terms which we cannot compute. More specifically, if we would want to calculate $\log p(x_n)$ directly we would have to compute the KL term $KL(q(Z|x_n)||p(Z|x_n))$ which contains the intractable integral $p(Z|x_n))$. So we cannot simply move the KL term on the left hand side of equation 14 to the right side to compute the log likelihood directly. Instead we have to optimize the lower bound i.e. the ELBO.

Question 1.5

From equation 14 we see that two things can happen if our ELBO increases (the lower bound is pushed up):

1. The KL term $KL(q(Z|x_n)||p(Z|x_n))$ becomes smaller, meaning our approximated posterior $q(Z|x_n)$ becomes closer to the actual intractable posterior $p(Z|x_n)$.
2. The log likelihood term $\log p(x_n)$ increases (the likelihood over our data distribution), which is the objective that we want to maximize.

Question 1.6

Reconstruction loss

The term reconstruction is appropriate because this error term computes the probability of the images reconstructed by our decoder given the latent representation Z that is sampled from the distribution learned by our encoder. Put differently, this term encourages the encoder to produce a latent representation that in turn causes the decoder to produce images that are "likely" or similar to real training images. So because of this error/loss term our VAE network needs to learn to generate and reconstruct images that are good (likely based on seen training data).

Regularization loss

The term regularization is appropriate because this positive error term adds a penalty to our loss function if the distribution returned by the encoder diverges from our prior, which is the standard normal distribution. I will explain why this is useful. We want our generated representation of the latent space to exhibit two important properties: 1) continuity, which means that points close in the latent space should not produce two totally different images once decoded and 2) completeness, which means that a point sampled from the latent space should produce a meaningful image once decoded.

If we do not apply a regularization term, the model can learn to "cheat" the learning objective by directly overfitting to reconstructing images that are similar the input data. In this case the encoder does not produce a meaningful Gaussian-like distribution from which we can sample new images but it can return distributions with tiny variances or return distributions with diverging means

(that are far apart in the latent space). So then the continuity and completeness of our latent space is not guaranteed.

To overcome this we apply the regularization term, which forces the learned distribution of our encoder to diverge not too much from the standard Gaussian. This way, we prevent our VAE network to cheat our objective of producing meaningful distributions over our latent space from which we can sample, thereby ensuring the strong property of VAEs which is that we can use them to sample meaningful new images.

Source for answering this question: [1].

Question 1.7

The steps we take in calculating our loss terms:

1. compute $\mu_\phi(x_n)$ and $\Sigma_{phi}(x_n)$ by inputting x_n to the encoder network
2. sample $z_n \sim \mathcal{N}(\mu_\phi(x_n), diag(\Sigma_\phi(x_n)))$
3. given our singly sampled latent variable z_n compute $\log p(x_n|Z)$ by approximating $\log p(x_n|Z) \approx \log p(x_n|z_n) = -\log \prod_{m=1}^{M} \text{Bern}(x_n^{(m)}|f_\theta(z_n)_m)$ (f_θ is our decoder neural network)
4. compute $\mathcal{L}_n^{\text{recon}} = -\mathbb{E}_{q_\phi(z|x_n)}[\log p_\theta(x_n|Z)] = -\log p(x_n|z_n)$. So $\mathcal{L}_n^{\text{recon}}$ is essentially the negative log likelihood we computed at step 3. This means we can use the Binary Cross Entropy loss function for calculating $\mathcal{L}_n^{\text{recon}}$ when implementing this in code.
5. We compute $\mathcal{L}_n^{\text{reg}}$, which is the KL divergence between two multivariate Gaussians. $\mathcal{L}_n^{\text{reg}} = \frac{1}{2}\big(tr(\Sigma_\phi(x_n)) + \mu_\phi(x_n)^T\mu_\phi(x_n) - D - \log\det|\Sigma_\phi(x_n)|\big)$ [2]

   Sidenote: the KL divergence between two multivariate Gaussians can also be computed by using the formula for the univariate Gaussians and subsequently summing over the D-dimension of the mean and variance - this is also how I applied it in my code.
6. We compute the total averaged loss objective $\mathcal{L} = \sum_{n=1}^{N} \mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}$

Question 1.8

In calculating $\nabla_\phi \mathcal{L}$ (i.e. the gradients with respect to the parameters of the encoder network) we would need to backpropagate throught the sampling process that happens between the encoder and decoder network:$z_n \sim q_\phi(z_n|x_n) = \mathcal{N}(\mu_\phi(x_n), diag(\Sigma_\phi(x_n)))$. This sampling operation is a non-continuous operation meaning it has no gradient and we cannot perform backpropagation through it [2].

The reparametrization trick allows us to perform backpropagation through the sampling operation by rewriting the random variable z_n as a differentiable transformation function in terms of fixed determenistic values for the mean and variance of our approximated distribution, where the stochasticity of the sampling operation is shifted by sampling another random (noise) variable from the standard normal distribution: $\epsilon \sim \mathcal{N}(0, 1)$. This gives the expression: $z_n = \mu_\phi(x_n) + \Sigma_\phi(x_n) \odot \epsilon$. So instead of the non-differentiable sampling operation $z_n \sim \mathcal{N}(\mu_\phi(x_n), diag(\Sigma_\phi(x_n)))$ we now have a differentiable function $z_n = \mu_\phi(x_n) + \Sigma_\phi(x_n) \odot \epsilon$ which is essentially the same operation but with a shift of stochasticity so that the functino is differentiable and we can perform backpropagation [2].

Question 1.9

For training the model I used the following hyperparameters:

Batch size: 128
Hidden dimensions: 512, 512
Act. function: ReLU
Learning rate: 1e-3
Epochs: 80

z_dim: 20
seed: 42

So mostly I just used the default hyperparameters, as training with these settings already produced decent results. I did add an extra hidden layer to my VAE network however, as I noticed that this did improve results. The training and validation bpd plots are displayed in figure 1 (note that unfortunately I could not combine the two plots into one diagram which would have been a bit nicer, since I created my diagrams using Tensorboard). The final test bpd score I obtained using this model configuration was:

$$\text{bpd}^{\text{test}} = 0.152$$

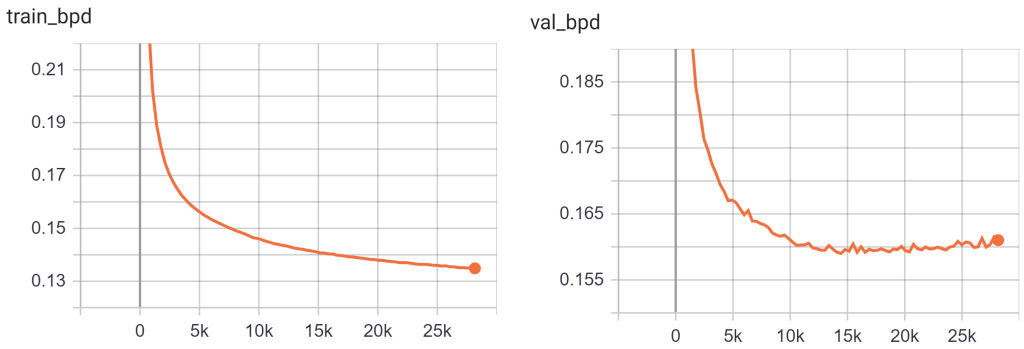


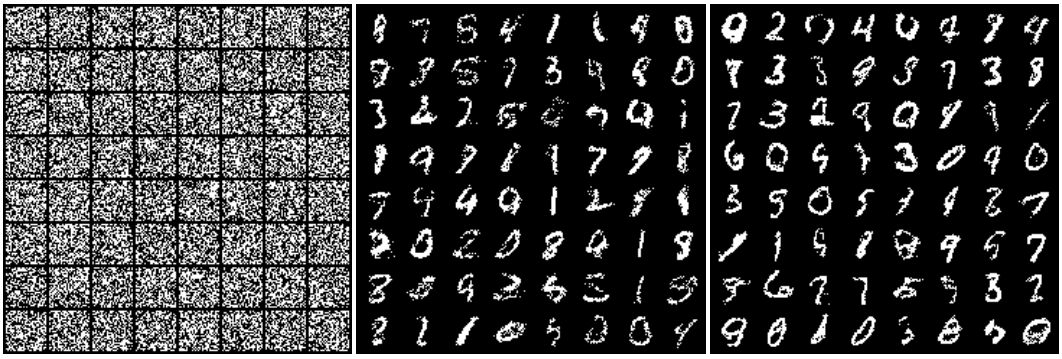
Figure 1: Training bits-per-dimension score **(a)** and validation bits-per-dimension score **(b)** plots throughout the training process of the VAE-MLP model (trained for 80 epochs with batch size 128). For both plots: y-axis = bpd score, x-axis = training step.

Question 1.10

64 binarized samples (8 × 8 grid) generated from the VAE model at three points throughout training are shown in figure 2. In this figure you can see that the model starts by generating random noise (fig. 2a) which is of course what you might expect. After 10 epochs however, the model already does a reasonable job in generating digit-like images. At epoch 80, almost all images are pretty clear in which digit they represent, although they are still a bit noisy. This noise is also due to the fact that I have plotted the binary sampled image. To illustrate how the continuous sampled images look like, I have also plotted these samples in figure 3. Here we notice much smoother images which are perhaps somewhat better readable but also blurrier than the binarized images. Moreover, I do not necessarily think there are really big problems with my generated images (the generations are not too noisy and they also vary enough so that every digit can be spotted in the images), the only problem is that some images are a bit ambiguous in what digit they represent. Altogether, I think the VAE model does a reasonable job in generating new images (although the MNIST dataset is naturally relatively simplistic compared to most real-life data).

Question 1.11

The visualized data manifold is displayed in figure 4. As expected, we the plot is similar to the plot in [3]. We see a smooth transition between the sampled images over the entire latent space. All images can be clearly recognized within the manifold. What is noticeable is that images that can be confused with each other are also placed near each other within the grid (e.g. a 4 can be interpreted as a 9 when poorly written, we can see this on the right-above side of the plot). This is not unsurprising, but it is fun to see how smoothly the decoder transits between digits and is able to return ambiguous digits when transitioning from one digit representation in the latent space to another.



(a) Epoch 0 **(b)** Epoch 10 **(c)** Epoch 80

Figure 2: Sampled images (**binarized**) from the VAE-MLP model at different stages throughout training (64 images per sample stage).

(a) Epoch 0 **(b)** Epoch 10 **(c)** Epoch 80

Figure 3: Sampled images (**continuous**) from the VAE-MLP model at different stages throughout training (64 images per sample stage).

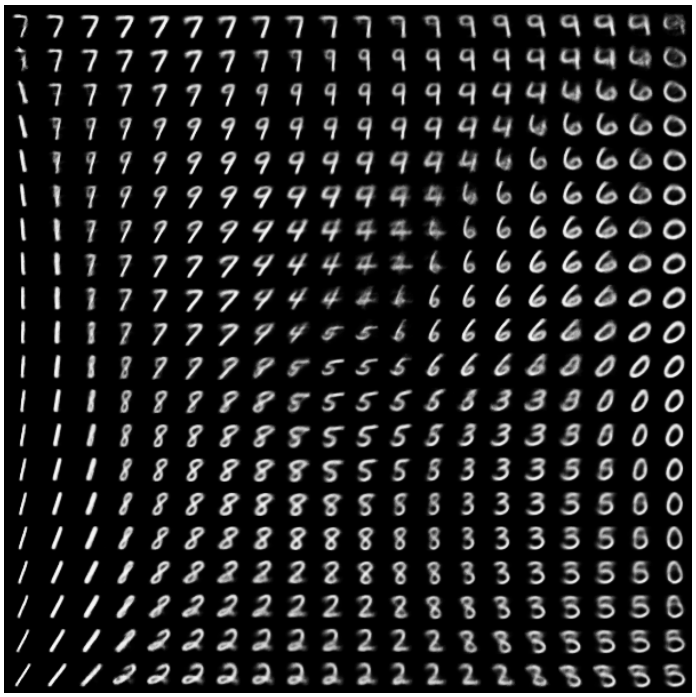


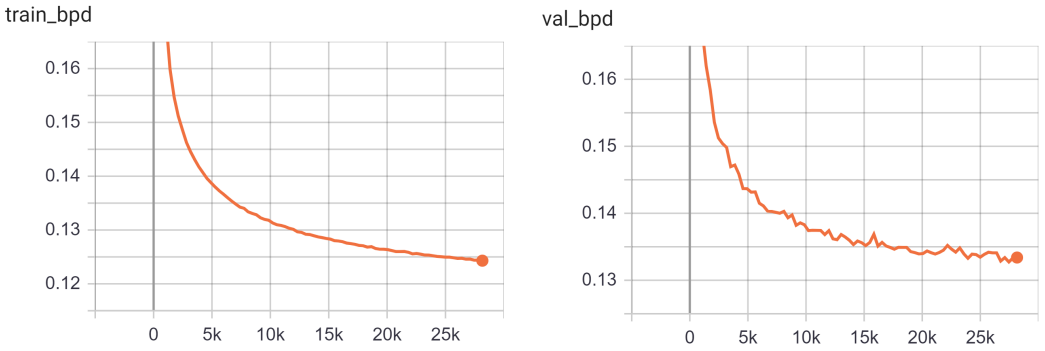
Figure 4: The data manifold visualized as the output images over the entire (high density) latent space.

Question 1.12 (BONUS) The training and validation bpd plots of the CNN-based VAE model (which I adapted from tutorial 9) are displayed in figure 5. The final test bpd score I obtained using this model configuration was:

$$\text{bpd}^{\text{test}} = 0.129$$

The binarized and continuous sampled images during different stages of training are displayed in figure 6 and 7, respectively.

When comparing to the MLP model we applied earlier, it is noticeable that the CNN model achieves better performance in terms of bpd score (final test score of 0.129 for the CNN versus a score of 0.152 for the MLP). What is more surprising however, is that the generated images of the CNN model actually look worse then the MLP model. They are a lot more noisy and "wildly written" compared to the images generated by the VAE-MLP model. One reason I can think of for this happening is that a CNN-based decoder model has trouble with assigning characteristics of digits (features that it recognizes) to spatial parts of images. The strong property of CNNs, the fact that they are so good in detecting features everywhere in images because of the weight-sharing filters that slide over images, is now actually something that limits them to reconstruct images. The deconvolution process in the decoder is thus a lot more noisy than the fully connected MLP decoder (so it seems).



(a) **(b)**

Figure 5: Training bits-per-dimension score **(a)** and validation bits-per-dimension score **(b)** plots throughout the training process of the VAE-CNN model (trained for 80 epochs with batch size 128). For both plots: y-axis = bpd score, x-axis = training step.

(a) Epoch 0 **(b)** Epoch 10 **(c)** Epoch 80

Figure 6: Sampled images (**binarized**) from the VAE-CNN model at different stages throughout training (64 images per sample stage).

(a) Epoch 0 **(b)** Epoch 10 **(c)** Epoch 80

Figure 7: Sampled images (**continuous**) from the VAE-CNN model at different stages throughout training (64 images per sample stage).

2 Generative Adversorial Networks

Question 2.1
a)

1. $\mathbb{E}_{p_{data}(x)}[\log D(X)]$ represents the expected log-probability of the discriminator correctly classifying real (training) images as real
2. $\mathbb{E}_{p_z(z)}[1 - \log D(G(Z))]$ represents the expected log-probability of the discriminator correctly classifying images generated by the generator (fake) images as fake.

In this minimax game, the discriminator D tries to maximize the objective (so maximizing these two terms), while the generator tries to minimize the objective (minimizing the two terms, thus it tries to 'fool' the discriminator).

b) Theoretically, after training converged (at the equilibrium) the generator would generate images that are indistinguishable from the real training images, so that the discriminator cannot do better than 'random guessing'. This means that the prediction of the generator becomes: $\text{p}^{\text{real}} = \text{p}^{\text{fake}} = 0.5$ and then we have:

$$V(D, G) = \log \frac{1}{2} + \log \frac{1}{2} = 2 \log(\frac{1}{2}) = -2 \log 2$$

Question 2.2

Early on during training, the generator will be quite bad at producing reasonable images (it starts by making random noise). Because the generator is so poor in the beginning and generated images are nothing like real training images, the discriminator has a very easy job and can easily reject all generated images as being fake. Because the generated then becomes too good, the term

$$1 - \log D(G(Z))$$

in the minimization objective of the generator becomes (very close to) zero, which means that there is no loss for the generator to use for backpropagation. So we then have the problem of vanishing gradients early on in training (the loss plateau of the generator becomes flat) which makes learning of the discriminator very hard.

A solution proposed to overcome this is the usage of the heuristic non-saturing loss [4]. Here the training objective for the discriminator stays the same, but the optimization objective for the generator becomes:

$$\max \ \mathbb{E}_{p_z(z)}[\log D(G(Z))]$$

So the generator tries to maximize the likelihood of the generator being mistaken on the fake images it creates. By doing this, the generator can learn even when the discriminator successfully rejects all generated images as being fake. Moreover, by simplifying the objective for the generator, i.e. it only looks at the behaviour of the discriminator when classifying generated images, it is not bothered when the discriminator is confused by real training images (which could lead to unexpected training behaviour).

Question 2.3

I used the following setup for my GAN implementation:

z_dim: 32
hidden_dims_generator: 128, 256, 512
hidden_dims_discriminator: 512, 256
dropout_rate_gen: 0.1
dropout_rate_discriminator: 0.3
learning rate: 2e-4
optimizer: Adam
betas: (0.5,0.999)
batch_size: 128
epochs: 250
seed: 42

So I did not adapt anything from the default parameters. The generated images throughout different stages of training are displayed in figure 8. Similar to the generated images of the VAE models in part 1, the GAN model starts by generating meaningless images. After 10 epochs of training, the generated images are still very noisy and not really good. After 250 epochs, however, the GAN model does generate images that can be easily identified as a certain digit in most samples. So similar to the VAE, the quality of the images improves over time and after a certain amount of training it does a decent job in generating new images. The VAE seemed to produce better images earlier on in training though, since the generated images of the VAE after 10 epochs of training are much better than those of the GAN after 10 epochs. There does not seem to be any mode collapse in the training process, as we see a varying range of different output images (digits).

(a) Epoch 0 **(b)** Epoch 10 **(c)** Epoch 250

Figure 8: Sampled images from the GAN network at different stages throughout training (64 images per sample stage).

Question 2.4

Four interpolations between two latent vectors are displayed in figure 9. What we can see from the figure is that the generator transitions between (the constructed image of) the latent vector left towards the (constructed image) of the latent vector on the right side (horizontally). What is really interesting here, is that the generator seems to generate totally different numbers when interpolating

between the two positions in our latent space. For example, in the first row we can see the generator producing a digit resembling 8 in the middle when interpolating between the two digits 3 (left) and 1 (right). In the second row, it generates a 3 when interpolating between a 0 and a 5. I noticed more of this hop-like behaviour when analyzing my results. So it seems the generator is able to make very sudden jumps in terms of output images when we move a bit in our latent space. These sudden jumps are then also some sort of mode changes, where we suddenly see a totally different number than in our previous interpolation step. The sudden changes in output images becomes also very clear when we compare figure 9 with the smooth-transitioning manifold in figure 4 that our VAE produced.

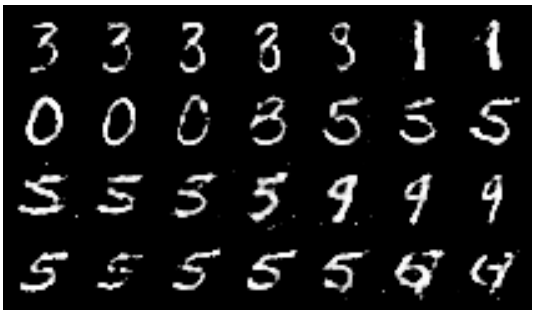


Figure 9: Four interpolations between two latent noise vectors, displayed as the constructed images by the generator (after 220 epochs, 5 interpolation steps **horizontally**).

Question 2.5

I did not experience any of the mentioned problems (at least, not severely). Mode collapse did not happen during my training process, and the network did converge, although I did have oscillations of my loss and accuracy for both the generator and the discriminator. Nevertheless, Non-convergence and in its severest form mode collapse, is a big problem in training GANs. The problem with mode collapse is that the generator converges to the distribution of one specific training image or class (e.g. it learns to perfectly generate a 0). Subsequently it only produces 0s until the discriminator simply starts rejecting all 0s it encounters. The generator then hops to another specific (point-like) distribution of another specific training example and this cat and mouse game continues until the end of training. This is undesired behaviour, as the diversity of generated samples will be very low. Although mode collapse is still a big problem in the realm of GANs it can be dampened by applying some regulariztion term to the generator which punishes the model when it produces images that are too similar too each other.

Also, the problem of training instability is frequent in the usage of GANs and I did experience slight instability during training. Techniques one could try to alleviate this problem to smoothen training could be to use label smoothing, (virtual) batch normalization and/or adding random noise to true training images. I would have gladly tried out these techniques on my own implementation, but due to time constraints I unfortunately was not able to.

3 Normalizing Flows

Question 3.1

$$
\begin{aligned}
z = f(x) &= x^3 \\
p_x(x) = p_z(z)|\frac{df(x)}{dx}| & \\
&= p_z(x^3)|\frac{df(x)}{dx}| \\
&= \frac{1}{b-a}|\frac{df(x)}{dx}|\mathbb{I}[a \leq x^3 \leq b] \\
&= \frac{1}{b^{1/3}-a^{1/3}}3x^2\mathbb{I}[a^{1/3} \leq x \leq b^{1/3}]
\end{aligned}
$$

Where $\mathbb{I}$ is the indicator function. So:

$$
\begin{aligned}
&\int_{-\infty}^{\infty} \frac{1}{b^{1/3}-a^{1/3}}3x^2\mathbb{I}[a^{1/3} \leq x \leq b^{1/3}] \\
&\quad = \frac{1}{b^{1/3}-a^{1/3}}(b^{1/3}-a^{1/3}) = 1
\end{aligned}
$$

Question 3.2

a) h_l and h_{l-1} need to be of the same dimensionality to ensure the mapping of f: $R^D \rightarrow R^D$. So $h_l \in R^D$.

b) The problem that could arise here is that computing the Jacobian of high-dimensional matrices is computationally very heavy and sometimes perhaps even practically intractable (it is generally a very complex computation). Computations and transformations then might take too much time to complete. So for normalizing flow models to work properly, especially when computational resources are scarce, we need to make sure that our transformation functions are easily invertible, differentiable and we must be able to efficiently compute the determinant of the jacobian **J**. So if a flow model is to be of any practical use, it is best to only apply transformations whose determinants of Jacobians are easy to compute, such as planar and radial flows.

Question 3.3.

Normalizing flows are continuous operations. If we apply flow models on discrete data such as images, the learned model will place probability mass on integer (pixel) values only. Moreover, the flow model will assign arbitrarly high δ-peak probabilities on these few discrete locations, so we have a sparse probability distribution with high peaks only at some discrete pixel values. This discrete modelling of probabilities is nonsensical and there is no smoothness, which is something that is normally a strong property of good probability distributions.
To fix this problem, we could apply some dequantization on the discrete input images so the data becomes continuous. An existing method to do is variational dequantization, where we add some continuous sampled noise $u \sim q(u|x)$ to the input data: $v = x + u$. Here, instead of using a standard uniform distribution or some standard Gaussian distribution, we could actually learn a good noise distribution $q(u|x)$ in a variational manner, hence the name variational dequantization.

b) The steps for training and validation are essentially the same except for the last step. The steps during training/validation boil down to density estimation in the forward direction and are:

1. Our input consists of the training images. First we apply dequantization on the input data to obtain continous input images.
2. Subsequently, we apply our series of transformation functions (e.g. coupling layers) on our input data x

3. After the last transformation layer we can compute the log probability $p_x(x)$ using equation 24 (so here we need to compute the log determinant of the Jacobian)
4. Now that we have our log likelihood we can maximize this by defining a loss function (i.e. the negative log likelihood) and we can subsequently backpropagate and update our transformation parameters (Note: we only apply this step when training, if we validate we can stop after the previous step)

If we want to sample new images from the flow model, we invert the process above to generate images, beginning at the latent space and moving backward by inverting the transformation functions. The steps:

1. Sample latent representations/vectors from our prior distribution p(z)
2. Move backwards trough the transformation functions (e.g. coupling layers)
3. Apply the inverse dequantization operation to obtain our generated image

4 Conclusion

Question 4.1

This report has covered VAEs, GANs and Flow-based models. All techniques are generative models, that can serve to generate new data that is similar to the images of the training data that they have seen. This can be useful for numerous cases, such as filling up missing data, create realistic simulations or even simulate future developments. However, all models approach this generation process in a different manner.

Variational Autoencoders (VAEs) essentially approximate the underlying distribution of our data (the data likelihood p(x)) by maximizing a lower bound (ELBO) of this distribution. Normalizing flows on the other hand, are able to directly model and optimize the data (log) likelihood by applying a series of transformations, which is therefore an inherent advantage of using flow models over VAEs. In practice, flow models are therefore also capable of generating new images of higher quality. However, whereas training and sampling of VAE models is generally very stable and fast, training flow models can become computationally heavy quickly if a lot of transformations are necessary to model the data distribution. Moreover, another difference between VAEs and flow models can be found in their modelling of the latent representation z: VAEs model the underlying distribution of the latent variables whereas flow models can map input data (e.g.) pixels directly to their respective latent value. GAN models work totally different in this matter, GAN models do not learn latent representations of our input data at all (they do not learn any likelihood distribution at all - so they do not model p(x)). Instead GANs work by directly generating images that look as real as possible by playing a minimax game of two players (a generator and a discriminator). An advantage of GANs is that they often produce the best images in practice, and the sampling process is fast and efficient. However, a downside is that no explicit likelihood distribution is learned when using GANS and more importantly, training GANs can be a hard process due to instability, non-convergence or mode collapse.

Altogether, all three methods bring their own advantages and disadvantages when compared to each other, and each of the methods still has its flaws which future research may solve. Perhaps a combination of these three methods will prove to be the eventual victor within the branch of generative models of Deep Learning, such as (already existing) architectures in which VAEs are combined with flow based models. Who knows, within the domain of Deep Learning everything can change within a period of 5 years...

References

[1] J. Rocca, "Understanding variational autoencoders (vaes)." https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73, 2019. accessed on: 08-12-2020.

[2] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[3] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.

[4] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," *arXiv preprint arXiv:1701.00160*, 2016.