Deep learning Assignment 3

Raymond Koopmanschap - 11925582
raymond.koopmanschap@student.uva.nl

1 Variational Auto Encoders

1.1 Latent variable models

Question 1.1

1. No, they both try to find out what the real distribution of the data is and want to minimize the reconstruction error. However the VAE also assumes for example a Gaussian as latent space and want to learn the parameters of that specific Gaussian. Where the normal autoencoder just sees this as a normal layer of a neural network, so just a vector representation of the data.
2. A normal autoencoder is not generative because the latent space is a fixed vector, you can't sample from it.
3. Depending on the task, A VAE can be used in place of a standard auto-encoder. It also tries to reconstruct your image or any other data that you have. But it can't generate new images.
4. The VAE learns a known distribution as latent space, for example a Gaussian. It learns the parameters of that Gaussian. By using a known distribution we can sample from it and thus generate new images.

1.2 Decoder: The Generative Part of the VAE

Question 1.2

Ancestral or forward sampling is using the probabilistic graph to sample from a joint distribution and go forward through this graph. Since x_n depends conditionally on z_n we will first sample from z_n. Then we can calculate the probability of f_θ and finally we can sample a x_n, m times.

Question 1.3

Before we sample x_n we transform z_n by a formula f_θ which can approximate arbitrarily complicated distributions given that it is expressive enough. Since we can choose f_θ anything we like, we don't really restrict ourselves.

Question 1.4

a. We can approximate $logp(x_n)$ by sampling from the distribution and take the average value of this samples.

$$\begin{aligned}\log p(x) &= \log \int p(x_n|z_n)p(z_n)dz_n \\ &= \log \mathbb{E}_{p(z_n)}[p(x_n|z_n)] \\ &\approx \log \frac{1}{k}\sum_{k=1}^{K} p(x_n|z_n^k)\end{aligned}$$

Where: $z_n^k \overset{i.i.d}{\sim} p(z_n)$

K is the total number of samples we take and z_n is sampled from the distribution $p(z_n)$. This converges due to the strong law of large numbers.

b. In order to provide a good approximation for $\log p(x)$ we need to cover roughly the whole space of the distribution $p(z_n)$, this becomes increasingly difficult for larger dimensionality of z. Image you can get a good estimate by picking n values in a 1D case, for the 2D case you will need n^2 values to cover the space equally well. This scales very bad and thus becomes very inefficient for higher dimensions.
Another reason is that for most samples of $p(z_n)$, the probability of $p(x_n|z_n)$ will be very small and thus not really contribute to a good approximation.

1.3 The Encoder

Question 1.5

a. Given 1, we can see that $\mu_q = 0$ and $\sigma_q = 1.1$ (or very close to 1), will give a very small value (0.064) for the KL-divergence. Chosing a very different mean, will result in a distribution that does not really overlap thus having a very large value for the KL-divergence, for example $\mu_q = 10$ and $\sigma_q = 1$, will result in a KL-divergence of 5.

b. The general formula is

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

Now filling in $p = \mathcal{N}(0, 1)$ gives

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

Question 1.6

$$\log p\left(\boldsymbol{x}_n\right) - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right) = \mathbb{E}_{q(z|\boldsymbol{x}_n)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p(Z)\right) \tag{2}$$

$$\log p\left(\boldsymbol{x}_n\right) = \mathbb{E}_{q(z|\boldsymbol{x}_n)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p(Z)\right) + D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right) \tag{3}$$

$$\text{Now since the KL is non-negative:} D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right) \geq 0 \text{ and thus} \tag{4}$$

$$\log p\left(\boldsymbol{x}_n\right) \geq \mathbb{E}_{q(z|\boldsymbol{x}_n)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p(Z)\right) \tag{5}$$

According to 5, the $\log p(x_n)$ is always larger than the right hand side. Thus it is lower bounded. In other words it will never become lower than the right hand side. If we increase the right hand side we are guaranteed to increase $\log p(x_n)$, therefore this is called the lower bound.

Question 1.7

The log-probability ($\log p(x_n)$) is impossible to compute directly, as was also stated in question 1.4, it involves an intractable integral that is also not possible to compute with the Monte Carlo sampling due to the bad scaling to higher dimensions.
This lower bound is easier to compute because it uses the distribution of the approximate posterior $q(z|x_n)$ which is a distribution that we can choose. Thus using the lower bound makes optimizing $\log p(x_n)$ tractable.

Question 1.8

First, the $\log p(x_n)$ can increase which is what we want.
Second the $D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right)$ can become lower, which means the approximate posterior $q(Z|x_n)$ is closer to the real posterior $p(Z|x_n)$, which is also what we want.

1.4 Specifying the encoder

Question 1.9

First we will discuss the reconstruction loss. We can minimize this loss by making $\log p_\theta(x_n|Z)$ be close to 1. In order to do this, the model needs to learn how it can obtain samples x_n with high

probability given Z. Z is the latent encoding, so given our encoding how can we reconstruct our sample, so that we get samples that are very likely. Therefore we can see this as a reconstruction loss. Next the regularization loss. We can minimize this loss by letting the approximate posterior $q_\phi(Z|x_n)$ be close to the prior distribution $p_\theta(Z)$. This can be seen as enforcing the latent space to have a certain distribution. By enforcing such a distribution the model is less likely to overfit on the training set. It will be less able to fit the distribution exactly like the dataset, making us more able to generalize better to unseen samples.

Question 1.10

We will rewrite the reconstruction loss using 1 sample.

$$\begin{aligned}
\mathcal{L}_n^{\text{recon}} &= -\mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta\left(x_n|Z\right)\right] && (6)\\
&\approx -\log p_\theta(x_n|z_n) && (7)\\
&= -\log \prod_{m=1}^{M} \text{Bern}\left(\boldsymbol{x}_n^{(m)}|f_\theta\left(\boldsymbol{z}_n\right)_m\right) && (8)\\
&= -\sum_{m=1}^{M} \log((f_\theta\left(\boldsymbol{z}_n\right)_m)^{x_n^{(m)}}(1-f_\theta\left(\boldsymbol{z}_n\right)_m)^{1-x_n^{(m)}}) && (9)\\
&= -\sum_{m=1}^{M} x_n^{(m)} f_\theta\left(\boldsymbol{z}_n\right)_m + (1-x_n^{(m)})(1-f_\theta\left(\boldsymbol{z}_n\right)_m) && (10)
\end{aligned}$$

Equation 10 represents the binary cross entropy loss. Next we will write out the regularization loss. The KL divergence for the multivariate gaussian case is given by

$$\frac{1}{2}\left[\log\frac{|\Sigma_2|}{|\Sigma_1|} - d + \text{tr}\left\{\Sigma_2^{-1}\Sigma_1\right\} + \left(\mu_2-\mu_1\right)^T\Sigma_2^{-1}\left(\mu_2-\mu_1\right)\right]$$

The derivation can be seen here derivation multivariate gaussian KL. Where tr() is the trace of the matrix,|A| indicates the determinant of matrix A and d is the number of dimensions.
Filling this in gives us

$$\begin{aligned}
\mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}\left(q_\phi\left(Z|x_n\right)\|p_\theta(Z)\right)\\
&= D_{\text{KL}}\left(\mathcal{N}(\mu_q,\Sigma_q)\|\mathcal{N}(0,I_d)\right)\\
&= \frac{1}{2}\left[\log\frac{1}{|\Sigma_q|} - d + tr\{\Sigma_q\} + \mu_q^T\mu_q\right]
\end{aligned}$$

With this we can compute the final loss $\mathcal{L}$.

1.5 The reparametrization trick

Question 1.11

(a) The approximate posterior $q_\phi(z_n|x_n)$ is a neural network and we would like to update the parameters ϕ. In order to do this we need to backpropagate through the network. Therefore we need to compute the gradient of the loss $\nabla_\phi\mathcal{L}$.

(b) We can't backpropagate through a sampling operation because we can't take the derivative of a sampling process. Therefore we can't update the parameters ϕ

(c) Instead of sampling from a Gaussian with mean μ and standard deviation σ, we sample from a standard Gaussian $\mathcal{N}(0,1)$ and shape this into our original Gaussian by using $\mathcal{N}(\mu,\sigma) = \mu + \sigma\mathcal{N}(0,1)$. Now we only have to sample from the $\mathcal{N}(0,1)$ Gaussian and this doesn't change. Since the σ and μ are now multiplication respectively addition, we can take the derivative and compute the gradients over the network.

1.6 Putting things together: Building a VAE

Question 1.12

The same structure as in (1) is used for the encoder and decoder. Only instead of tanh activations, ReLU's are used because that gave slightly better performance. Thus a linear layer with ReLU and

two seperate linear layers for both the mean and log variance. The decoder exist of a linear layer, ReLU, linear, sigmoid. The reconstruction loss is implemented as the binary cross entropy, the only difference is that we take the sum of all pixel values (pytorch takes the average). The regularization loss is implemented as in (1) appendix C.

Question 1.13

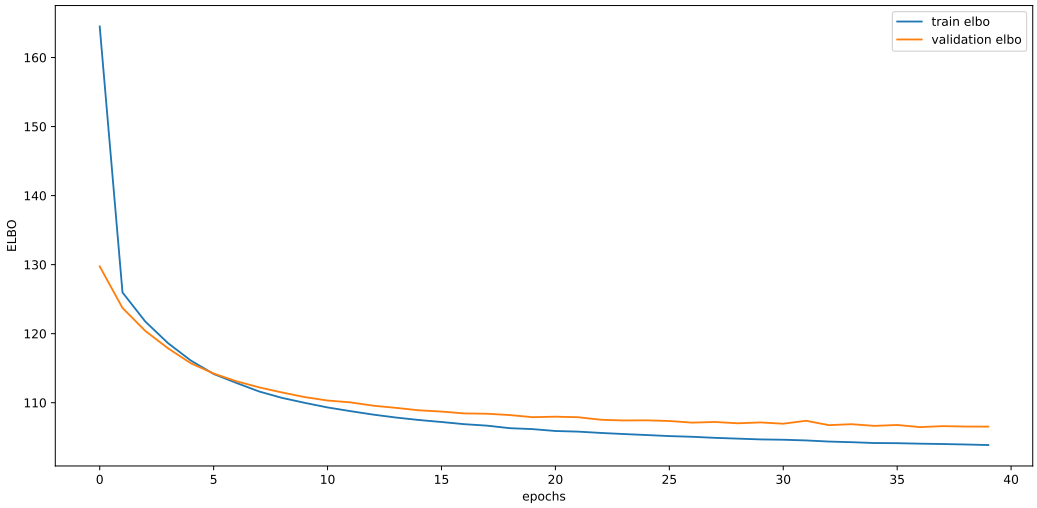
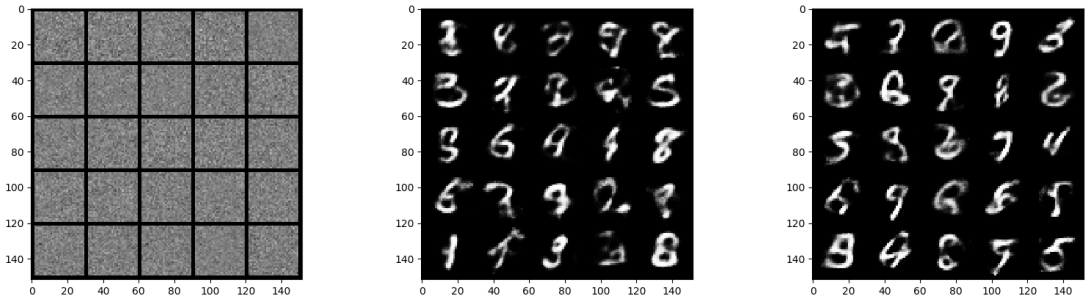


Figure 1: Negative ELBO for 40 epochs

The negative ELBO should start around the value 784 * $\log(0.5) \approx 543$. With a random seed of 42 it starts at 549, which is close enough. The difference can be explained by random initialization. Note that 1 start after the first epoch, this is to obtain a smoother plot.

Question 1.14



(a) Epoch 0 (b) Epoch 20 (c) Epoch 40

Figure 2: Bernoulli means after epoch 0, 20 and 40

There is a small improvement but the model doesn't seem to learn a lot anymore after epoch 20. Also the negative ELBO doesn't decrease a lot anymore, so this could make sense.

Question 1.15

The variation in 3 is not really high. It gets higher over time. First it seems to learn 1 and 9, later on 7 and other numbers. Note that for example 4, seems a too complex number for the 2 dimensional latent dimension so it can not be found in the manifold.

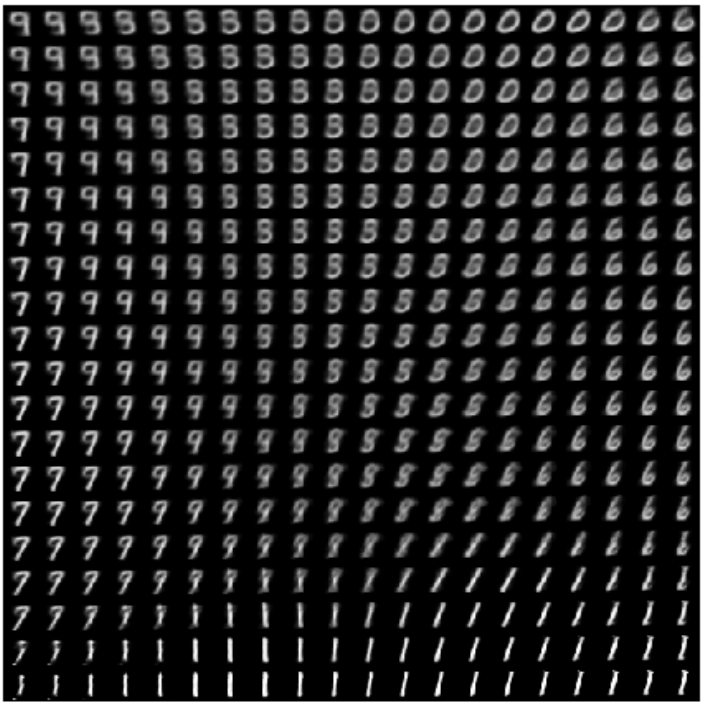


Figure 3: The manifold with 20 rows and 20 columns

2 Generative Adversarial Networks

Question 2.1

Discriminator

- Input: Feature vector x, in case of a 32x32 image this is a feature vector of length 32*32 = 1024. Note that the input doesn't have to be an image.
- Output: A probability that indicates the class label. A value close to 1 means the system is very certain it is an image from the real dataset and a value close to 0 means it thinks that it is a fake image (generated by the generator).

Generator

- Input: A noise vector, for example random variables sampled from a standard Gaussian distribution could be used, but other distributions are also possible.
- Output: A feature vector that has the same dimensions of the input of the discriminator, so it can be fed into the discriminator. This output feature vector could for example represent a 32x32 image in the case that the discriminator has images of 32x32 as input.

2.1 Training objective: A Minimax Game

Question 2.2

The first term:

$$\mathbb{E}_{p_{\text{data}}(x)}[\log D(X)]$$

Indicates the expected value of the log-likelihood of the image x coming from the real dataset, where x is sampled from the real data distribution. The discriminator wants to maximize this, since it wants

to output values close to 1, if the image comes from the real dataset. It wants to be as close to the real data distribution as possible.
The second term is:

$$\mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))] \tag{11}$$

Which we will break down in smaller components. First $D(G(Z))$ means that the output of the generator is fed into the discriminator. The discriminator wants to classify this close to 0, since it is a fake image. Now 11 means the expected value of the log-likelihood of $1 - D(G(Z))$, where z is sampled from the noise distribution that goes into the generator. Now the discriminator wants to minimize the term $D(G(Z))$ and thus maximizing $log(1 - D(G(Z)))$. Thus the discriminator wants to correctly classify fake images as fake. The generator on the other hand is the opponent of the discriminator and wants exactly the opposite hence the minimum of the described two terms.

Question 2.3

We will try to find the optimal solution of the discriminator by maximizing equation 15 of the assignment for the discriminator.

$$V(D, G) = \mathbb{E}_{p_{\text{data}}(x)}[\log D^*(X)] + \mathbb{E}_{p_z(z)}[\log(1 - D^*(X))] \tag{12}$$

$$= \int_x p_r(x) \log(D(x)) + p_g(x) \log(1 - D(x))dx \tag{13}$$

$$= p_r(x) \log(D(x)) + p_g(x) \log(1 - D(x)) \tag{14}$$

The last step can be done because there is sampled over all x. Now the optimal discriminator is given by $\frac{dV}{dD} = 0$, since we want to find a maximum.

$$\begin{aligned}
\frac{dV}{dD} &= \frac{p_r(x)}{D(x)} - \frac{p_g(x)}{1 - D(x)} = 0 \\
\frac{p_r(x)}{D(x)} &= \frac{p_g(x)}{1 - D(x)} \\
p_r(x)(1 - D(x)) &= p_g(x)D(x) \\
D(x) &= \frac{p_r(x)}{p_r(x) + p_g(x)}
\end{aligned}$$

When the learning converges, the generator generates perfect samples that are very close to the real distribution, thus $p_g(x) \to p_r(x)$ and therefore $D^*(x) = \frac{1}{2}$. Filling this into equation **??** we obtain

$$\begin{aligned}
V(D^*, G^*) &= \int_x p_r(x) \log \frac{1}{2} + p_g(x) \log \frac{1}{2} dx \\
&= \log \frac{1}{2} (\int_x p_r(x) + \int_x p_g(x)) \\
&= \log \frac{1}{2}(1 + 1) \\
&= -2 \log 2
\end{aligned}$$

Question 2.4

At the beginning the generator and discriminator are both not very good, however in most cases the discriminator will quickly become pretty good. If the discriminator is able to detect the still bad fake images of the generator, then the term $\log(1 - D(G(Z)))$ will become 0 and thus there will be no loss and thus no gradients, therefore the generator can't learn and will not become better. To solve this problem the non-saturing hueristic loss is used for the generator. So instead of minimizing $1 - D(G(Z))$ we will maximize

$$\log D(G(Z)) \tag{15}$$

This objective is in principal the same, but it will give better gradients early on in the training phase.

2.2 Building a GAN

Question 2.5

The generator and discriminator are constructed with the recommended layers. Only a dropout layer with probability 0.25 after each leaky ReLU is added in the discriminator, this gave quite a lot better performance.
The procedure is as follows: First noise is generated and fed into the generator. Next we fed those images into the discriminator, the generator want to make those images real, so label is 1. This makes sure we obtain 15 as loss function.
The second step is training the discriminator, the generated images are fed into the discriminator and he tries to classify those images as fake, label is 0 and we also feed the real images and he tries to classify those images as real, label is 1. The combined loss of those is the total loss of the discriminator and based on that an update step is performed.

Question 2.6

(a) Epoch 0 (b) Epoch 100 (c) Epoch 200

Figure 4: Generated images after epoch 0, 100 and 200

Question 2.7

Figure 5: Interpolated images

3 Generative Normalizing Flows

3.1 Change of variables for Neural Networks

Question 3.1

The only thing that changes is that we replace the derivative with the determinant of the Jacobian.

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

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

Question 3.2

First of all f needs to be invertible, which was already mentioned in the previous exercise. Next the dimensions of x and z has to be the same. Because the flow maps one distribution to another only in the same dimension.

Question 3.3

The determinant of the Jacobian can take a very long time to compute. In general when the Jacobian is a n by n matrix, it takes $\mathcal{O}(n^3)$ time to compute. This scales very bad with large dimensions and takes too much time. Therefore we need to make sure that the Jacobian only has non-zero diagonal entries or that it is triangular (only non-zero at the lower or upper triangle).
Another issue is that f needs to be easily invertible. Using a sigmoid as non-linear layer in neural networks can lead to unstable solutions because of the sigmoid saturating and therefore a small change in the invertible input can have a large change in the output.

Question 3.4

When using discrete data, the continuous density model will put all probability mass on those discrete datapoints, leading to a sub-optimal solution since neural networks or other function approximators are not good in finding good solutions on uniform distributions. A solution to this problem is dequantisizing the discrete distribution into a continuous distribution. In earlier work this was done by adding uniform noise between each datapoint, thus in general for unit hypercubes of dimension D, as the space between pixel values (1, 2 ... 255), is one. (**?**) improves on this by replacing the uniform noise with a distribution $q(u|x)$, where $u \in [0, 1]^D$. This distribution is a flow-based model which is also optimized, thus obtaining a more flexible distribution than the original uniform distribution.

3.2 Building a flow-based model

Question 3.5

During training you try to map the distribution of the data to noise (for example in the form of a Gaussian distribution). The input will be a datapoint sampled from the original distribution and the output will be a $\log$ probability. During inference time you use the inverted function to sample from the noise distribution and it will give back a datapoint that is hopefully likely under the original data distribution.

Question 3.6

1. Sample a batch
2. Normalize and dequantisize the samples in the batch
3. Split/Mask the data into two parts
4. The first set of data (1:d) is kept the same, while the second part (d+1:D) is going through the neural networks s and t and the output will be as specified in (**?**) equation 7:

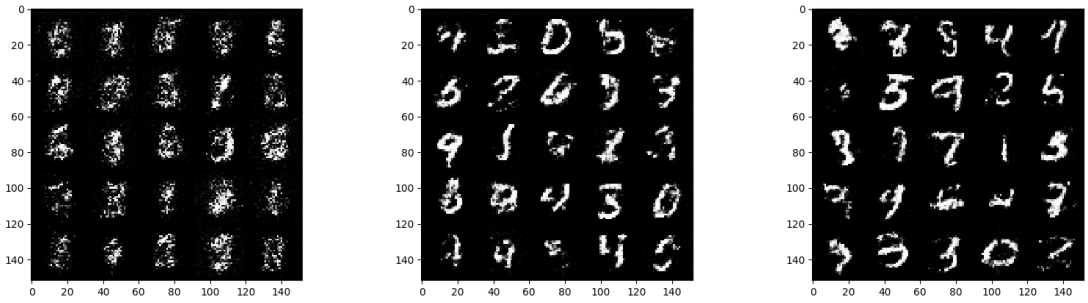
$$\begin{aligned} y_{1:d} &= x_{1:d} \\ y_{d+1:D} &= x_{d+1:D} \odot \exp\left(s\left(x_{1:d}\right)\right) + t\left(x_{1:d}\right) \end{aligned}$$

5. Calculate the log-likelihood for calculating the cost
6. Backpropagate through the network to update the weights

Question 3.7

Below are some results from the flow-based model.

The quality is less than the GAN and comparable to the VAE.



(a) Epoch 1 (b) Epoch 20 (c) Epoch 40

Figure 6: Images from the flow model after epoch 1, 20 and 40

Question 3.8

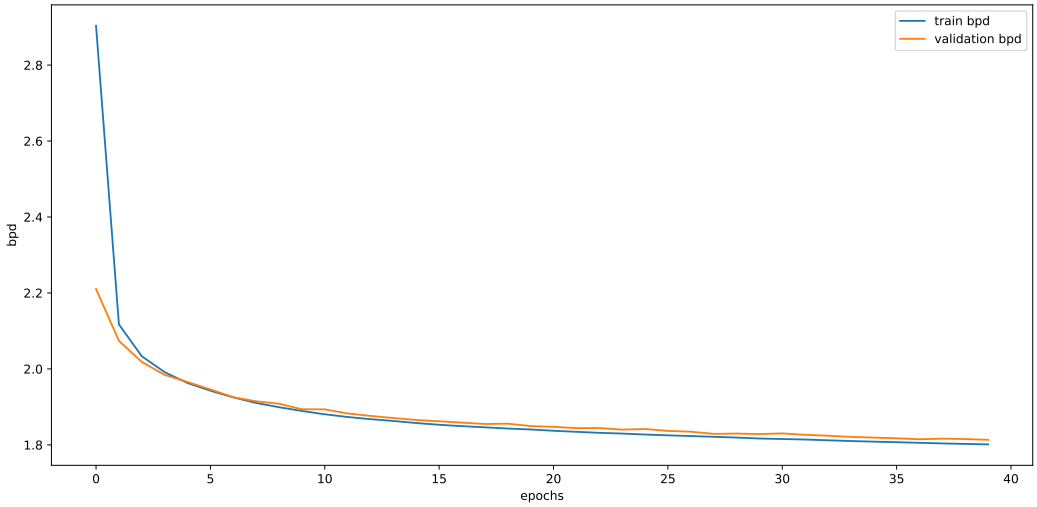


Figure 7: Bits per dimension performance for 40 epochs

4 Conclusion

Question 4.1

We have used 3 generative methods to generate realistically looking images. The GAN has the best performance and VAEs and Flow models roughly had the same performance, which was slightly worse than GANs. However a notable disadvantage of the GAN model is that the training can be unstable due to the generator and discriminator competing against each other. In terms of computational speed is the VAE the best model, running in less than an half hour on CPU and already converged (the model didn't really improve anymore). The flow-based model took around half an hour on GPU and the GANs 1.5 hours. The GPU's roughly decreases the computational time by 5 fold.
The GAN is mostly used for only generating images and the density is implicitly used. This in contrary to VAEs where the density is explicitly modeled by approximating it. The flow-model instead makes the density tractable by using change of variables and creating a computable Jacobian.

References

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