Deep Learning Assignment 3: Deep Generative Models

Elias Kassapis
12409782
University of Amsterdam
kassapiselias@hotmail.co.uk

Introduction

1 Variational Auto Encoders

1.1 Latent Variable Models

Question 1.1

1. Standard autoencoders and variational autoencoders (VAEs) have a similar structure, as they both consist of an encoder, a decoder, and a loss function. However, standard autoencoders learn a compressed representation of the input by learning a mapping of the input to the latent space, whereas VAEs learn the model of a distribution representing the input data. Therefore, these have different loss functions, were standard autoencoders use a distance loss function to optimize the mapping by quantifying the reconstruction loss, such as mean squares error, while VAEs make use of the expected cross-entropy loss (in the form of negative log-likelihood) with respect to the encoder's distribution over the representations, and a Kullback-Leibler divergence regularizer, to optimize the learned model distribution.

2. A standard autoencoder is not generative as it learns a one-to-one mapping. It's sole purpose is to compress the input in a latent representation and then decompress it to reconstruct the original input, for example to remove noise. Therefore, any variation between outputs from an autoencoder given the same input will be due to having a bad model. However, a VAEs are considered generative because they learns to generalize within the boundaries of the distribution over the factors of variation of the data, and they capture the dependencies between. Therefore, we can use VAEs to generate, as their names suggest, variable "legal" data samples.

3. A VAE could be used in place of a standard autoencoder, as we could reduce the standard deviation of the trained model to ensure only the maximum likelihood sample is returned instead of the whole distribution. I would expect this to return a reconstruction of the original image.

4. VAEs are able to generalize because in contrast to the standard autoencoder, we learn a distribution over latent variables underlying the intrinsic distribution of the input instead of only learning the fixed values (maximum likelihood) of these latent variables.

1.2 Decoder: The Generative Part of the VAE

Question 1.2

We have the following model for the pixels of the images $\mathbf{x}_n$ in our dataset

$$p\left(\mathbf{z}_n\right) = \mathcal{N}\left(0, \mathbb{I}_D\right) \tag{1}$$

$$p\left(\mathbf{x}_n|\mathbf{z}_n\right) = \prod_{m=1}^{M} \text{Bern}\left(\mathbf{x}_n^{(m)}|f_\theta\left(\mathbf{z}_n\right)_m\right) \tag{2}$$

were $\mathbf{z}_n$ is a latent variable of our statistical model, $\mathbb{I}$ is the identity matrix, D denotes the dimensionality of the latent space, $\mathbf{x}_n^{(m)}$ is the m_{th} pixel of the n_{th} image in our dataset $\mathcal{D}$, and $f_\theta : \mathbb{R}^D \rightarrow [0,1]^M$ is a neural network (the decoder of the VAE) with parameters θ that outputs the mean of the Bernoulli distributions for each pixel in $\mathbf{x}_n$.

In order to sample from such a model, we have to first draw samples of $\mathbf{z}_n$ from the multivariate Gaussian probability distribution $p\left(\mathbf{z}_n\right)$, and then map these to $\mathbf{x}_n$ by sampling M pixels $\mathbf{x}_n^{(m)}$ from a Bernoulli distribution parametrized by the output of $f_\theta\left(\mathbf{z}_n\right)_m$.

Question 1.3

The assumption that our latent variables follow a standard-normal distribution is not very restrictive in practice because using a set of d latent variables that are normally distributed, we can generate any distribution in d dimensions if we learn a sufficiently complicated function (such as a multi-layered neural network) to map the set of the d latent variables to the values required for the model. Therefore, in this case, we can use a multi-layered neural network as the decoder for our VAE, which learns a mapping f_θ from our latent variables $\mathbf{z}_n$ to a distribution over the parameter values for a Bernoulli distribution. Therefore, we learn the parameters for the function $f_\theta\left(\mathbf{z}_n\right)_m$ that maps set of M sampled latent variables $\mathbf{z}_n$, to the M required Bernoulli distributions, were each Bernoulli distribution represents a pixel, so that we obtain a model of an image consisting of M pixels.

Question 1.4

(a) The log probability of the data $\mathcal{D}$ under our model is given by

$$\begin{aligned}
\log p(\mathcal{D}) &= \sum_{n=1}^{N} \log p\left(\mathbf{x}_n\right) \\
&= \sum_{n=1}^{N} \log \int p\left(\mathbf{x}_n|\mathbf{z}_n\right) p\left(\mathbf{z}_n\right) \quad d\mathbf{z}_n \\
&= \sum_{n=1}^{N} \log \mathbb{E}_{p(z_n)}\left[p\left(\mathbf{x}_n|\mathbf{z}_n\right)\right]
\end{aligned}$$

Now evaluating the integral $\int p\left(\mathbf{x}_n|\mathbf{z}_n\right) p\left(\mathbf{z}_n\right) d\mathbf{z}_n$ is intractable to do analytically due to the shear number of computations required, because the dimensionality of $\mathbf{x}$ and $\mathbf{z}$ can be large , and we need to evaluate it over all possible configurations of the latent variables. Therefore, we can instead approximate $\log p(\mathcal{D})$ using Monte Carlo sampling to estimate $\log p\left(\mathbf{x}_n\right) = \log \mathbb{E}_{p(\mathbf{z}_n)}\left[p\left(\mathbf{x}_n|\mathbf{z}_n\right)\right]$. This is done by drawing K samples from $p\left(\mathbf{z}_n\right)$, using these to evaluate $p\left(\mathbf{x}_n|\mathbf{z}_k\right)$, and taking the mean of the values obtained. This is given by

$$\log p(\mathbf{x}_n) \approx \log\left(\frac{1}{K}\sum_{i=1}^{K} p(\mathbf{x}_n|\mathbf{z}_i)\right)$$

therefore
$\log p(\mathcal{D}) \approx \sum_{n=1}^{N} \log\left(\frac{1}{K}\sum_{i=1}^{K} p(\mathbf{x}_n|\mathbf{z}_i)\right)$

As we increase the number of samples our estimate becomes more accurate because our prior distribution $p(\mathbf{z}_n)$ in the integral $\int p(\mathbf{x}_n|\mathbf{z}_n)\, p(\mathbf{z}_n)\, d\mathbf{z}_n$ is absorbed in the summation in Equation (3), and is now represented by the frequency of the different $\mathbf{z}_n$ values sampled, therefore, the greater K is, the more the 'evidence' we have, which improves our posterior distribution $p(\mathbf{x}_n|\mathbf{z}_i)$, and therefore our approximation for $p(\mathbf{x}_n)$.

(b) This approach is inefficient because the reliability of our estimate of $p(\mathbf{x}_n|\mathbf{z}_i)$, depends on the number of samples of $\mathbf{z}_n$ we draw from $p(\mathbf{z})$. As the dimensionality of $\mathbf{z}$ increases, the number of samples required for an accurate approximation scales exponentially, and we can potentially need an extremely large number of samples for a highly dimensional $\mathbf{z}$. This is because for most $\mathbf{z}_n$ values sampled from the prior $p(\mathbf{z}_n)$, $p(\mathbf{x}_n|\mathbf{z}_n)$ is almost zero, as most values of $\mathbf{z}_n$) do not produce $\mathbf{x}_n$, as illustrated in Figure 1.

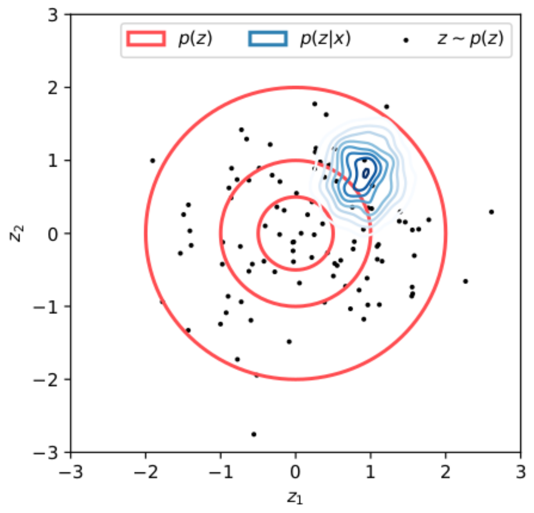

Figure 1: Plot of 2-dimensional latent space and contours of prior and posterior distributions.

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

Question 1.5

(a) KL-divergence is given by

$$D_{\mathrm{KL}}(q\|p) = -\mathbb{E}_{q(x)}\left[\log\frac{p(X)}{q(X)}\right] = -\int q(x)\left[\log\frac{p(x)}{q(x)}\right] dx$$

where q and p are probability distributions in the space of some random variable X. KL-divergence is essentially a distance measure between two probability distributions calculated by the weighted average on the difference between those distributions at X. When we are looking at the KL divergence between two univariate Gaussian distributions, such as q and p, the difference between the distributions at X depends on the means μ_q

and μ_p, and the variances σ_q^2 and σ_p^2. The closer the means and variances are between the two distributions, the higher their similarity. Therefore, since we have $p = \mathcal{N}(0,1)$, if $\left(\mu_q, \sigma_q^2\right) = (0 \pm c, 1 \pm d)$, where c and d are arbitrary constants with value close to zero (eg. $0 < c \leq 1$ and $0 < d \leq 1$), we would have a small KL divergence value, and if c and d are large (i.e. $c > 1$ and $d > 1$), we would have higher KL divergence. Therefore, an example of a very small divergence would be something like $\left(\mu_q, \sigma_q^2\right) = (0.1, 0.9)$, and an example of a very large KL divergence $\left(\mu_q, \sigma_q^2\right) = (10, 9)$.

(b) The closed-form KL-divergence of two univariate Gaussians q and p is given by

$$\begin{aligned} D_{\mathrm{KL}}(q \| p) &= -\int p(x) \log q(x) \;\; dx + \int p(x) \log p(x) \;\; dx \\ &= \frac{1}{2} \log \left(2\pi\sigma_p^2\right) + \frac{\sigma_q^2 + \left(\mu_q - \mu_p\right)^2}{2\sigma_p^2} - \frac{1}{2}\left(1 + \log 2\pi\sigma_q^2\right) \\ &= \log \frac{\sigma_p}{\sigma_q} + \frac{\sigma_q^2 + \left(\mu_q - \mu_p\right)^2}{2\sigma_p^2} - \frac{1}{2} \end{aligned}$$

were $q = \mathcal{N}\left(\mu_q, \sigma_q^2\right)$ and $p = \mathcal{N}\left(\mu_p, \sigma_p^2\right)$. Since we are given that $\mu_p = 0$ and $\sigma_q^2 = 1$, substituting these in we get

$$\begin{aligned} D_{\mathrm{KL}}(q \| p) &= \log \frac{(\sqrt{1})}{\sigma_q} + \frac{\sigma_q^2 + \left(\mu_q - (0)\right)^2}{2(1)} - \frac{1}{2} \\ &= \log \frac{1}{\sigma_q} + \frac{\sigma_q^2 + \mu_q^2}{2} - \frac{1}{2} \end{aligned}$$

Question 1.6

We have

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

By Jensen's inequality, the Kullback-Leibler divergence is always greater or equal to zero. Therefore, for the left-hand-side of Equation (4) we can infer that

$$\log p\left(\mathbf{x}_n\right) - D_{\mathrm{KL}}\left(q\left(Z|\mathbf{x}_n\right) \| p\left(Z|\mathbf{x}_n\right)\right) \leq \log p\left(\mathbf{x}_n\right)$$

Thus

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

Hence, the right-hand-side of Equation (4) can be interpreted as the lower bound on the log probability.

Question 1.7

We optimize the lower-bound instead of optimizing the log-probability directly because, $p(Z|\mathbf{x}_n)$ appears in the KL divergence in the left-hand-side of Equation (4), which depends on $p(\mathbf{x}_n)$. As we have seen, the latter is given by

$$p(\mathbf{x}_n) = \int p(\mathbf{x}_n|\mathbf{z}_n)\, p(\mathbf{z}_n) \quad d\mathbf{z}_n$$

Evaluating this integral analytically is intractable because it requires exponential time to compute, as discussed in Question 1.4.

Question 1.8

As we can see in Equation (4), the lower bound is pushed up when $\mathbb{E}_{q(z|\mathbf{x}_n)}[\log p(\mathbf{x}_n|Z)]$ is maximized, and $D_{\text{KL}}(q(Z|\mathbf{x}_n)\,\|p(Z))$ is minimized. This is equivalent to maximizing $\log p(\mathbf{x}_n)$, and minimizing $D_{\text{KL}}(q(Z|\mathbf{x}_n)\,\|p(Z|\mathbf{x}_n))$. Therefore, we don't need to compute and minimize the KL divergence between our approximate and the exact posteriors, which is intractable as explained in Question 1.7. Instead we minimize the KL divergence between our approximate posterior and our prior and maximize the expectation of $\log p(\mathbf{x}_n|Z)$ with respect to our approximate posterior. The latter indirectly trains our encoder (our approximate posterior) to be more similar to the exact posterior, and the former prevents it from being too similar.

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

Question 1.9

The loss for our model, re-written in terms of per-sample losses is given by

$$\begin{aligned}\mathcal{L}(\theta,\phi) &= -\frac{1}{N}\sum_{n=1}^{N}\mathbb{E}_{q_\phi(z|\mathbf{x}_n)}[\log p_\theta(\mathbf{x}_n|Z)] - D_{KL}\left(q_\phi(Z|\mathbf{x}_n)\,\|p_\theta(Z)\right)\\ &= \frac{1}{N}\sum_{n=1}^{N}\left(\mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}\right)\end{aligned}$$

where

$$\begin{aligned}\mathcal{L}_n^{\text{recon}} &= -\mathbb{E}_{q_\phi(z|\mathbf{x}_n)}[\log p_\theta(\mathbf{x}_n|Z)]\\ \mathcal{L}_n^{\text{reg}} &= D_{\text{KL}}\left(q_\phi(Z|\mathbf{x}_n)\,\|p_\theta(Z)\right)\end{aligned}$$

Now

- $\mathcal{L}_n^{\text{recon}}$ is interpreted as the **reconstruction loss term** because the negative log likelihood is equivalent to the cross-entropy error, which increases if the decoder's output does not reconstruct the data well, therefore it encourages the model to improve reconstruction.

- $\mathcal{L}_n^{\text{reg}}$ is interpreted as a **regularization term** because this measures how much our encoder $q_\phi(z_n|x_n)$ differs from our multivariate Gaussian prior $p(\mathbf{z}_n) = \mathcal{N}(0, \mathbb{I}_D)$, so that the more these differ, the greater our loss. Therefore it penalizes the encoder outputs representations z_n that differ than those from a standard normal distribution, ensuring that our estimated posterior does not over-fit on the input data, maintaining diversity.

Question 1.10

Our final objective to minimize is

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

Now, for $\mathcal{L}_n^{\text{recon}}$, we have

$$\mathcal{L}_n^{\text{recon}} = -\mathbb{E}_{q_\phi(z|\mathbf{x}_n)} \left[\log p_\theta \left(\mathbf{x}_n|Z\right)\right]$$

From Equation (2) we know that

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

therefore, the explicit form of $\mathcal{L}_n^{\text{recon}}$ is given by

$$\begin{aligned}
\mathcal{L}_n^{\text{recon}} &= -\mathbb{E}_{q_\phi(z|\mathbf{x}_n)}\left[\log \prod_{m=1}^{M} \text{Bern}\left(\mathbf{x}_n^{(m)}|f_\theta\left(\mathbf{z}_n\right)_m\right)\right] \\
&= -\mathbb{E}_{q_\phi(z|\mathbf{x}_n)}\left[\log \prod_{m=1}^{M} \left(f_\theta\left(\mathbf{z}_n\right)_m\right)^{\mathbf{x}_n}\left(1 - f_\theta\left(\mathbf{z}_n\right)_m\right)^{1-\mathbf{x}_n}\right] \\
&= -\mathbb{E}_{q_\phi(z|\mathbf{x}_n)}\left[\sum_{m=1}^{M} \mathbf{x}_n \log\left(f_\theta\left(\mathbf{z}_n\right)_m\right) + \left(1-\mathbf{x}_n\right)\log\left(1 - f_\theta\left(\mathbf{z}_n\right)_m\right)\right]
\end{aligned}$$

where $f_\theta\left(\mathbf{z}_n\right)$ is pixel m of the reconstructed image, and $\mathbb{E}_{q_\phi(z|\mathbf{x}_n)}$ refers to averaging over the number of samples from the encoder (During SGD it is common practice to sample only one z per iteration). Therefore, $\mathcal{L}_n^{\text{recon}}$ is equivalent to binary Cross Entropy (not normalized over the pixel dimension).

For $\mathcal{L}_n^{\text{reg}}$, we have

$$\mathcal{L}_n^{\text{reg}} = D_{\text{KL}}\left(q_\phi\left(Z|\mathbf{x}_n\right) \| p_\theta(Z)\right)$$

Now, the closed form KL-divergence for two multivariate Gaussians q and p is given by

$$D_{\text{KL}}(q\|p) = \frac{1}{2}\left[\log\frac{\det\{\Sigma_p\}}{\det\{\Sigma_q\}} - d + \text{tr}\{\Sigma_p^{-1}\Sigma_q\} + (\mu_p - \mu_q)^T\Sigma_p^{-1}(\mu_p - \mu_q)\right]$$

were Σ_q and Σ_p are the covariance matrices for distribution q and p, respectively, and d denotes the dimensionality of the variable modelled.

From Equation (1) and Equation we know that

$$p\left(\mathbf{z}_n\right) = \mathcal{N}\left(0, \mathbb{I}_D\right)$$

We also know that

$$q_\phi\left(z_n | x_n\right) = \mathcal{N}\left(z_n | \mu_\phi\left(x_n\right), \operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)\right)$$

where μ_ϕ and Σ_ϕ are deterministic factors with parameters ϕ, learned from data, where $\mu_\phi(\cdot)$ maps an input image to the mean of the multivariate normal over $\mathbf{z}_n$, and $\operatorname{diag}(\Sigma_\phi(\cdot))$ maps the input image to the diagonal of the covariance matrix of that same distribution. Putting these together, the explicit form of $\mathcal{L}_n^{\text{reg}}$ is given by

$$\begin{aligned}
\mathcal{L}_n^{\text{reg}} =& D_{\text{KL}}\left(q_\phi\left(Z|\mathbf{x}_n\right) \| p_\theta(Z)\right) \\
=& \frac{1}{2}\left[\log \frac{\det\{(\mathbb{I}_D)\}}{\det\{\left(\operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)\right)\}} - D + \operatorname{tr}\{(\mathbb{I}_D)^{-1}\left(\operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)\right)\}\right] \\
&+ \frac{1}{2}\left[\left((\mathbf{0}) - \left(\mu_\phi\left(x_n\right)\right)\right)^T (\mathbb{I}_D)^{-1} \left((\mathbf{0}) - \left(\mu_\phi\left(x_n\right)\right)\right]\right. \\
=& \frac{1}{2}\left[\operatorname{tr}\left(\operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)\right) - D + \left(\mu_\phi\left(x_n\right)\right)^T\left(\mu_\phi\left(x_n\right)\right) - \log\det\{\operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)\}\right]
\end{aligned}$$

1.5 The Reparametrization Trick

Question 1.11

(a) We need $\nabla_\phi \mathcal{L}$ because we want to learn the parameters for our decoder $q_\phi\left(z_n | x_n\right)$ that allow us to produce codes for $\mathbf{x}_n$ that our decoder can reliably decode.

(b) We cannot train our model using Stochastic Gradient Decent (SGD) by back-propagating the error through the sampling nodes of the encoder, because a sample has fixed value, and therefore it's gradient is 0. As a result, the encoder does not receive a gradient signal, and does not learn anything. Hence, we implement out sampling of z_n from outside the network, allowing us to backpropagate through the decoder.

(c) The reparametrization trick is instead of sampling from the encoder, we first sample $\epsilon \sim \mathcal{N}(0, \mathbb{I})$, and then compute $\mathbf{z}_n$ by translating and scaling ϵ using $\mathbf{z}_n = \mu_\phi\left(x_n\right) + \operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)^{\frac{1}{2}} \times \epsilon$. Therefore, our objective function becomes

$$\mathbb{E}_{x_n \sim \mathcal{D}}\left\{\mathbb{E}_{q_\phi(z|\mathbf{x}_n)}\left[\log p_\theta\left(\mathbf{x}_n | Z = \mu_\phi\left(x_n\right) + \operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)^{\frac{1}{2}} \times \epsilon\right)\right] - D_{KL}\left(q_\phi\left(Z|\mathbf{x}_n\right) \| p_\theta(Z)\right)\right\}$$

Hence, we can now backpropagate and update $\mu_\phi\left(x_n\right)$ and $\operatorname{diag}\left(\Sigma_\phi\left(x_n\right)\right)$, that is, the mean and variance for our decoder $q_\phi\left(z_n | x_n\right)$.

1.6 Putting things together: Building a VAE

Question 1.12

For this part of the assignment, I implemented a VAE in PyTorch, and trained it for 100 epochs using ADAMs optimizer on the Binary MNIST data provided, with randomly initialized weights. I set my encoder to learn the log variance instead of the covariance, and modified the reparametrization trick, shown in Question 1.11, to convert log covariance back to covariance. Using the trick, I drew single samples of $\mathbf{z}_n$ for each training iteration. I used the objective function derived in Question 1.10, but I modified the $\mathcal{L}_n^{\text{reg}}$ term, again, converting log covariance back to covariance, to facilitate log variance learning. The intuition of this is that even if the learned log variance is negative, taking the exp of log variance will return a positive covariance. Finally, I found that normalizing both $\mathcal{L}_n^{\text{reg}}$ and $\mathcal{L}_n^{\text{rec}}$ over the batch and pixel dimensions gave the best results. Therefore, the latter is now standard Binary Cross Entropy. The architecture of my model is shown for a batch size of 1 in Table 1.

Layers	**Channels In**	**Channels Out**
Linear 1	784	500
ReLU	500	500
Dropout	500	500
Linear 2a	500	20
Linear 2b	500	20
Linear 3	20	500
ReLU	500	500
Dropout	500	500
Linear 4	500	784
Sigmoid	784	784

Table 1: Architecture details of the VAE implementation. The layers Linear 1 to Linear 2b make up the Encoder, and the rest the Decoder. Linear 2a outputs the learned mean $\mu_\phi(\mathbf{x}_n)$, and Linear 2b outputs the learned log covariance matrix $\log\{\text{diag}(\Sigma_\phi(\mathbf{x}_n))\}$ for the Encoder $q_\phi(\mathbf{z}_n|\mathbf{x}_n)$.

Question 1.13

The average negative Evidence Lower Bound (ELBO) was evaluated for every epoch using a 20-dimensional latent space, and the resulting curves for the training and validation sets are shown below in Figure 2.

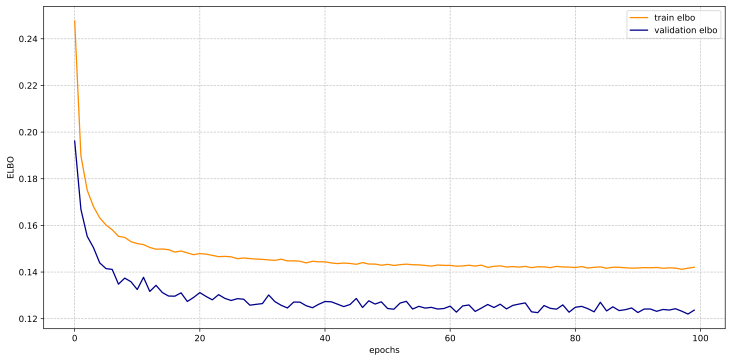

Figure 2: The average negative ELBO over every epoch.

As we can see, the negative ELBO decreases, indicating that the lower bound increases during training. Therefore, we get closer and closer to the ground truth $p(\mathbf{x}_n)$ distribution.

Question 1.14

For qualitative evaluation of my model, I sampled 25 20-dimensional latent vectors $\mathbf{z}_n$ from a standard normal distribution, and passed them through the decoder to generate the image means for each vector. These are shown below in Figure 3.

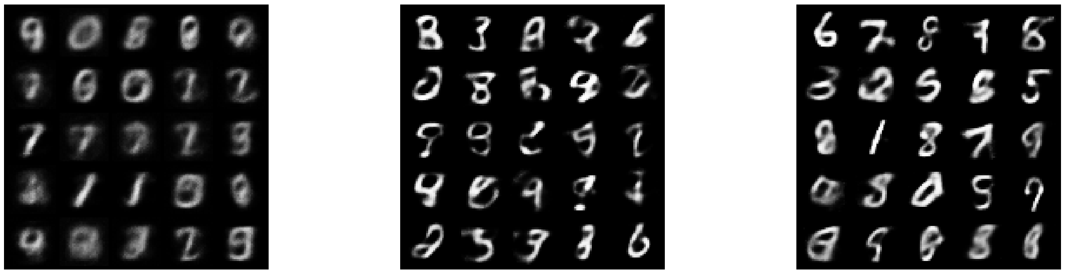

Figure 3: Plotted samples for epoch 1 (left), epoch 20 (center) and epoch 100 (right).

As we can see, the quality of the images generated improve as the number of training epochs increases, and the numbers become increasingly crisp as the model becomes more certain/confident over the approximated distribution of the ground truth (as the ELBO increases or negative ELBO decreases). The difference between the generated images from epoch 20 and epoch 100 are not as great as between those of epoch 1 and epoch 20. As we can see this correlates with the average negative ELBO plot depicted above in Figure 2.

Question 1.15

For this part of the assignment, I trained a VAE with a 2-dimensional latent space, by changing Linear layers 2a and 2b (shown in Table 1) to have 2 output channels instead of 20, and Linear layer 3 t have 2 input channels instead of 20. Everything else was kept the same. To visualize the manifold, I created a function that uses the percent point function (ppf) to create a two dimensional grid of points over the Z-space covering the parts that have significant density, passed the points through the decoder and plotted the results. These are illustrated below in Figure 4.

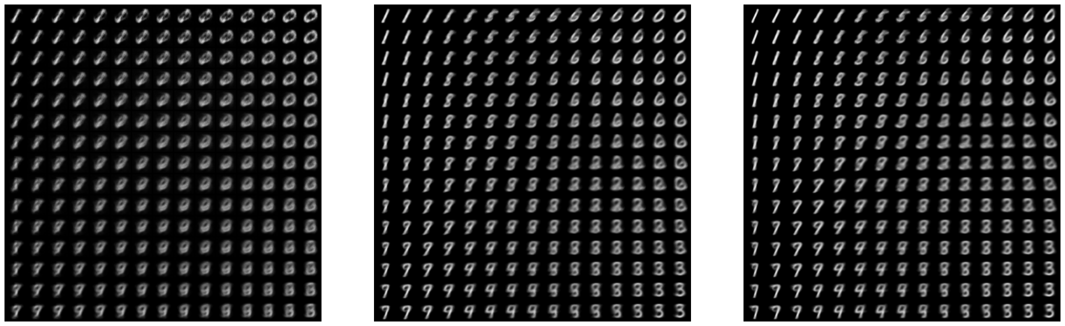

Figure 4: Plotted manifold for epoch 0 (left), epoch 20 (center) and epoch 100 (right).

As we can see, the manifold includes more numbers as training elapses, indicating that the model learns a better approximation of the ground truth distribution. Also, the quality of the generated numbers when we use a latent dimensionality of 2 is much better than the quality of those generated when a latent dimensionality of 20 is used (comparing the digits in the images here with the digits in the images in Figure 4). The improved representation when using a latent space with lower dimensionality is probably because the dimensions better capture the factors of variation of the particular data (binary minst), where a space of higher dimensionality may increase the degrees of freedom, therefore making it more difficult to control for the dependencies between the pixels.

2 Generative Adversarial Networks

Question 2.1

Vanilla Generative Adversarial Networks (GANs) consist of two networks, the Generator network and the Discriminator network. The input to the Generator network is a vector of random noise **z**, sampled from a normal or uniform distribution, which is up-sampled to output an image, whereas the Discriminator network is a standard convolutional network, which gets an image as an input, and classifies the image as fake or real by outputting a value between 0 and 1.

2.1 Training objective: A Minimax Game

Question 2.2

The Vanilla GAN objective function is given by

$$\min_G \max_D V(D,G) = \min_G \max_D \ \mathbb{E}_{p_{\text{ data}}(\mathbf{x})}[\log D(\mathbf{X})] + \mathbb{E}_{p_z(\mathbf{z})}[\log(1 - D(G(\mathbf{Z})))] \quad (4)$$

The first term of Equation (5), $\mathbb{E}_{p_{\text{ data}}(\mathbf{x})}[\log D(\mathbf{X})]$, represents the ability of the Discriminator to recognize real images, and the second term, $\mathbb{E}_{p_z(\mathbf{z})}[\log(1 - D(G(\mathbf{Z})))]$, represents the ability of the Discriminator to recognize fake, generated images. The objective of the Discriminator is to maximize $V(D,G)$; that is, its ability to discriminate well between real and fake images, and thus, maximizing both terms, whereas the Generator's objective is to minimize $V(D,G)$ by minimizing the ability of the Discriminator to recognize generated images, therefore, only affects the second term.

Question 2.3

After training has converged, the value f $V(D,G)$ is the minimax value of the Discriminator output with respect to the Discriminator and Generator network. This happens when the probability distribution learned by the generator $p_z(\mathbf{z})$ is the same as the probability distribution underlying the real samples p_{data}; that is, when $p_z(\mathbf{z}) = p_{data}$. Now let $p_z(\mathbf{z}) = p_{\text{gen}}$. Then we have

$$\begin{aligned} V(D,G) &= \mathbb{E}_{p_{\text{data}}(\mathbf{x})}[\log D(X)] + \mathbb{E}_{p_z(z)}[\log(1-D(G(Z)))] \\ &= \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{x}\sim p_{\text{gen}}}[\log(1-D(\mathbf{x}))] \\ &= \int_{\mathbf{x}} \left(p_{\text{ data }}(\mathbf{x})\log(D(\mathbf{x})) + p_{\text{gen}}(\mathbf{x})\log(1-D(\mathbf{x}))\right)\, d\mathbf{x} \end{aligned} \tag{5}$$

Now the equation in the integral is of the form

$$f(x) = a\log(\tilde{x}) + b\log(1-\tilde{x})$$

where $a = p_{\text{ data }}(\mathbf{x})$, $b = p_{\text{gen}}(\mathbf{x})$, and $\tilde{x} = D(\mathbf{x})$. The optimal value of $f(x)$ is given by $\frac{\partial f(x)}{\partial x} = 0$; that is

$$\begin{aligned} \frac{\partial f(x)}{\partial x} &= \frac{a}{\tilde{x}} - \frac{b}{1-\tilde{x}} = 0 \\ \frac{a}{\tilde{x}} &= \frac{b}{1-\tilde{x}} \\ a(1-\tilde{x}) &= b(\tilde{x}) \\ a - a(\tilde{x}) &= b(\tilde{x}) \\ a &= a(\tilde{x}) + b(\tilde{x}) \\ a &= (a+b)\tilde{x} \\ \tilde{x}^* &= \frac{a}{a+b} \end{aligned}$$

Therefore

$$D^*(\mathbf{x}) = \frac{p_{\text{ data }}(\mathbf{x})}{p_{\text{ data }}(\mathbf{x}) + p_{\text{gen}}(\mathbf{x})}$$

So when $p_{\text{gen}} = p_{data}$, we get $D^*(\mathbf{x}) = \frac{1}{2}$. Substituting this in Equation (6) gives

$$\begin{aligned} V(D^*,G^*) &= \int_{\mathbf{x}} \left(p_{\text{ data }}(\mathbf{x})\log\left(\frac{1}{2}\right) + p_{\text{gen}}(\mathbf{x})\log\left(1-\left(\frac{1}{2}\right)\right)\right)\, d\mathbf{x} \\ &= \log\left(\frac{1}{2}\right)\int_{\mathbf{x}} p_{\text{ data }}(\mathbf{x})\, d\mathbf{x} + \log\left(\frac{1}{2}\right)\int_{\mathbf{x}} p_{\text{gen}}(\mathbf{x})\, d\mathbf{x} \\ &= \log\left(\frac{1}{2}\right)\left(\int_{\mathbf{x}} p_{\text{ data }}(\mathbf{x})\, d\mathbf{x} + \int_{\mathbf{x}} p_{\text{gen}}(\mathbf{x})\, d\mathbf{x}\right) \\ &= 2\log\left(\frac{1}{2}\right) \end{aligned}$$

Hence the value of $V(D, G)$ after training has converged is $-2\log(2)$.

Question 2.4

In practice, optimizing the Generator network through it's objective function $J(G) = \mathbb{E}_{p_z(\mathbf{z})}[\log(1 - D(G(\mathbf{z})))]$ can be problematic because the slope of the gradient for $\log(1 - D(G(\mathbf{z})))$ is very saturated for low values of $D(G(\mathbf{z}))$ as illustrated by the blue line (we are using the Minimax version of $J(G)$) in Figure 5.

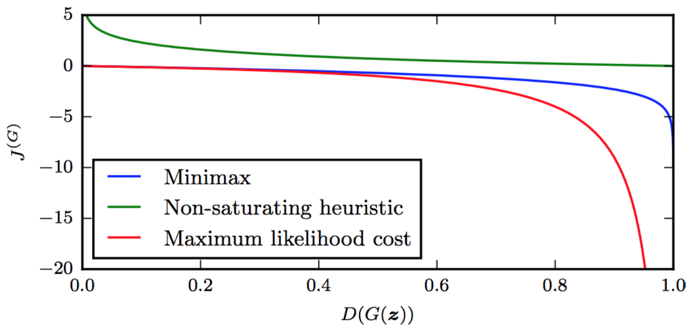

Figure 5: The gradient for different versions of the Generator network training objective $J(G)$ with respect to $D(G(\mathbf{z}))$. Taken from Seita (2017)

As we can see, our version of $J(G)$ at the initial stages of training; that is, when our Generator is not good enough to fool our Discriminator, returns a very weak gradient signal, but a high gradient diagonal when our Generator is already good ($D(G(\mathbf{z})) \approx 1$). The result is that our Discriminator network dominates the Generator network early on, $J(G)$ becomes stuck to low values of $D(G(\mathbf{z}))$, so the Generator only receives a weak gradient signal during training, and training becomes very slow or jammed.

A solution to this problem is to modify $J(G)$ in order to change the slope of its gradient to improve the gradient signal during training. For example, instead of minimizing the likelihood that the Discriminator is correct ($-log(D(G(\mathbf{z})))$), we can maximize the likelihood that the Discriminator is wrong ($log(D(G(\mathbf{z})))$), so $J(G) = \mathbb{E}_{p_z(\mathbf{z})}[\log(D(G(\mathbf{z})))]$. Then the gradient signal during early stages of the training will improve, preventing the Discriminator from dominating the Generator. This is shown by the green line in Figure 5, referred to as the non-saturated heuristc.

2.2 Building a GAN

Question 2.5

For this part of the assignment, I designed and implemented a GAN, using the recommended architecture, but for both the Generator and Discriminator, I added dropout (0.2) after each non-linearity. For the generator I used a Tanh last layer, and for the Discriminator, I used a Sigmoid last layer. Furthermore, I used the non-saturated heuristic shown in Q 2.4 as my loss function for the Generator, and the normal loss function for my Discriminator. Finally, I used an Adams optimizer, with parameters beta1 = 0.5 and beta2 = 0.999 as recommended by the InfoGAN paper (Chen et al., 2016). I trained the model for 200 epochs, and computed the average epoch loss for the Generator and for the Discriminator to get the loss curves depicted below in Figure 6.

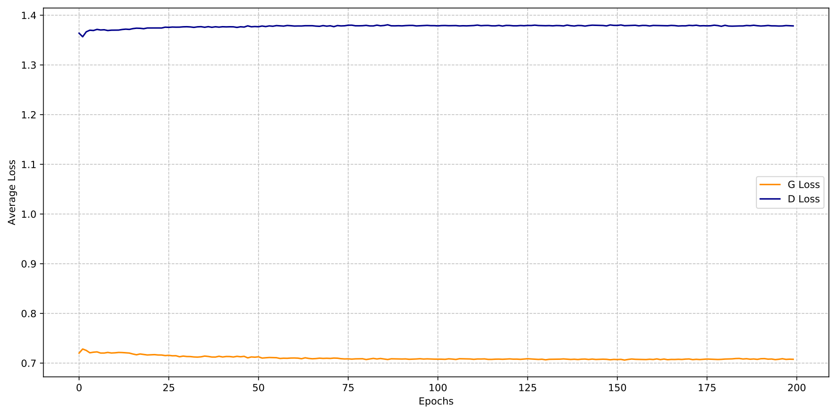

Figure 6: The average epoch loss for the Generator and Discriminator of my GAN implementation, over epochs elapsed

As we can see from the loss curves above, the loss of the Discriminator is higher than the Loss of the Generator, and the two loss curves appear to be reflections of each other, and almost parallel, illustrating that the relative "strength" of the generator and discriminator remains surprisingly stable throughout training. This indicates that either they don't learn anything, or they improve at the same rate, which is in fact the case as we will see below. The optimal value for the loss of the Discriminator is 0.5 as we have derived before, so this may be an indication that the Discriminator is not good enough, or the generator is too good. Perhaps removing the dropout layers or decreasing the dropout percentage from the discriminator or the generator to achieve this, however, I did not have enough time to test this, and my generated images looked fine.

Question 2.6

To qualitatively asses the performance of my GAN implementation, I sampled 25 noise vectors **z** from a standard Gaussian distribution, and propagated the noise through my Generator at different stages of the training. The results are shown below in Figure 7.

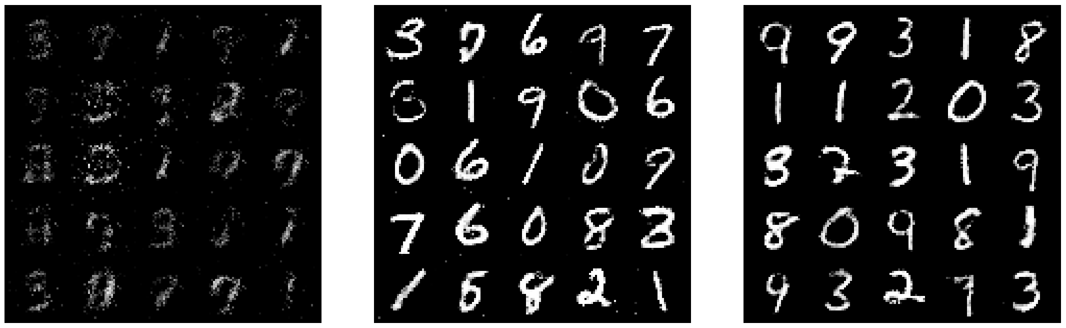

Figure 7: Plotted samples for epoch 0 (left), epoch 100 (center) and epoch 200 (right).

As we can see, as training time elapses, the samples increase in quality, indicating that the Generator is successfully trained. The samples at epoch 100 look comparable to those at epoch 200, indicating that my GAN implementation is already well trained at this point. The fact that the quality of the generated images is so good shows that both the generator and the discriminator do what they are supposed to do, as the discriminator is good enough to make the generator learn, but not too good, preventing the generator for learning, and the generator's quality is evident from the results.

Question 2.7

In this part of the assignment, I illustrate the quality of my model by interpolating between two digits in the latent space. This is done by setting generating noise vectors $\mathbf{z}$ with the same value for all dimensions, and interpolating between values within a set range by using a fixed interval for each interpolations step. As we can see this allows my to smoothly interpolate between the digit 7 to the digit 9 in 7 interpolation steps. Results are depicted in figure 8.

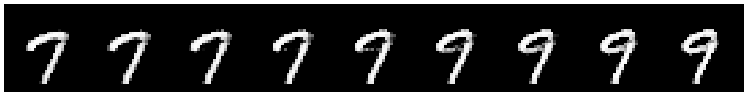

Figure 8: Interpolating between two digits in latent space.

This illustrates that we have smooth transitions between different classes in the latent space, although we did not explicitly enforce it, illustrating the power of GANs.

3 Generative Normalizing Flows

3.1 Change of variables for Neural Networks

Question 3.1

We have

$$z = f(x); \quad x = f^{-1}(z); \quad p(x) = p(z)\left|\frac{df}{dx}\right| \tag{6}$$

Therefore

$$\log p(x) = \log p(z) + \sum_{l=1}^{L} \log\left|\frac{dh_l}{dh_{l-1}}\right| \tag{7}$$

were L is the total number of variable changes, and h_l is the variable at change/step l. Now rewriting Equations (6) and (7) for the case when $f : \mathbb{R}^M \rightarrow \mathbb{R}^M$ is an invertible smooth mapping and $\mathbf{x} \in \mathbb{R}^M$ is a multivariate random variable gives

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

were $\det\{\cdot\}$ is the Jacobian determinant of it's input function (i.e. $\det\left\{\frac{df}{d\mathbf{z}}\right\}$ is the Jacobian determinant for function f).

Question 3.2

In order for this equation to be computable, the number of dimensions of $\mathbf{x}$, $\mathbf{z}$, and all variables changed in between ($\mathbf{h}_l$) should match (i.e. $f : \mathbb{R}^M \to \mathbb{R}^M$), because the Jacobian determinant is only defined for square Jacobian matrices. Also, by the inverse function theorem, in order for f to be invertible, the Jacobian determinant for f must not be zero, therefore this should also hold for all variables $\mathbf{h}_l$ for $0 < l < L$, where $\mathbf{h}_0$ = $\mathbf{x}$ and $\mathbf{h}_L$ = $\mathbf{z}$

Question 3.3

When optimizing our network using the objective derived in Question 3.1, f may not be easily invertible, and the Jacobian determinant of f may not be easy to compute. Addressing the latter, for invertible neural networks, the complexity for the Jacobian determinant's computation, and for the computation of it's derivative, typically scales as $O(LM^3)$, where L is the number of hidden layers (no. of variable changes), and M is the size of the hidden layers (dimensionality of the intermittent variables) (Rezende and Mohamed, 2015). Therefore, to ensure that the inverse and the Jacobian determinant of our functions are tractable, we want to carefully choose a family of transformations that preserves a low computational complexity for both operations.

Question 3.4

The consequence of storing images as discrete integers is that fitting a continuous density model on discrete representations of the data will place all the probability mass under the observed, discrete data points, leading to a very sparse, and jagged distribution, which is basically a mixture of point masses on discrete data. The result is that the fitted continuous density model cannot generalize, and is not an accurate representation of our data, as images in the real world are continuous signals. This problem could be fixed by converting the quantized, discrete representations of the images in our data into a continuous distribution (Ho et al., 2019). For example, our density model could be dequantized by adding uniform noise to the data over the width of each discrete bin to fill in the gaps between the discrete point masses of our density model. That is, if each dimension m (pixel) of image $\mathbf{x}$ takes on a value of one of the discrete bins given in $\{0, 1, 2, \ldots, 255\}$, we dequantize our data using $\mathbf{y} = \mathbf{x} + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ is sampled uniformly from $[0, 1)^M$, and M is the dimensionality of $\mathbf{x}$. Alternatively, we can introducing a dequantization noise distribution computed using variational inference to improve mass deposition between the discrete point masses of our density model.

3.2 Building a flow-based model

Question 3.5

During training, the input of the flow-based model is an M dimensional vector, for example a vector representing a real image $\mathbf{x}$, where M is the total number of pixels (as discussed above), and the output is a generated M dimensional vector sampled from p($\mathbf{x}$), such as a generated image $\mathbf{x}'$. Now, during inference time, the input is an M dimensional vector $\mathbf{z}$ sampled from the latent prior distribution $p(\mathbf{z})$, and the output is an M-dimensional vector representing a generated image $\mathbf{x}'$, which is generated by inverse mapping of $\mathbf{z}$ to $\mathbf{x}'$.

Question 3.6

During training, we follow the following steps:

1. we input mini-batches of real images **x**, each stretched out in a vector of size M (M = no. of pixels), and propagate the input through forward flow functions h_l, where each layer of the network l is implements one transformation, to get the latent vector **z**.

2. In this process, we compute the sum over the Jacobian determinants of each transformation, and use it to transform our prior probability distribution over **z** (which is a standard gaussian) to the probability distribution of p(**x**).

3. Then negative log of our approximation for p(**x**) is then used as our loss function and we back-propagate to update our intermediate transformation function parameters to improve our approximation.

Now, during inference time

1. We first sample a vector **z** from our prior distribution $p(\mathbf{z})$

2. We pass it though the reverse flow functions h_l^{-1}, to get the inverse mapping from **z** to **x**. (we do not backpropagate)

Question 3.7

For this part of the assignment, I implemented a flow-based model, inspired from the Real NVM model. I used the recommended architecture given to us by the assignment, and implemented the coupling layer as described in Dinh et al. (2017). I trained the model for 40 epochs using Adam optimizer with default settings.

Question 3.8

The loss curves of the training and validation set are shown below in Figure 10.

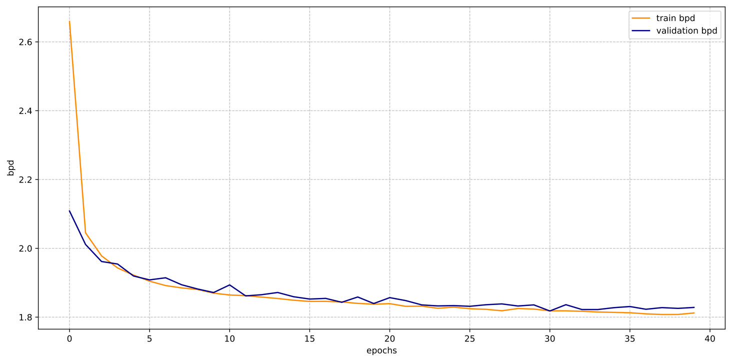

Figure 9: The average epoch loss for my NFs implementation, over epochs elapsed

As we can see, the model learns quickly with respect to number of epochs, however, the training time per epoch takes much longer than the other models. The model converges to a loss value of approximately 1.80 (to 2 d.f.). Below we can images generated as described in Q 3.6, in the inference section.

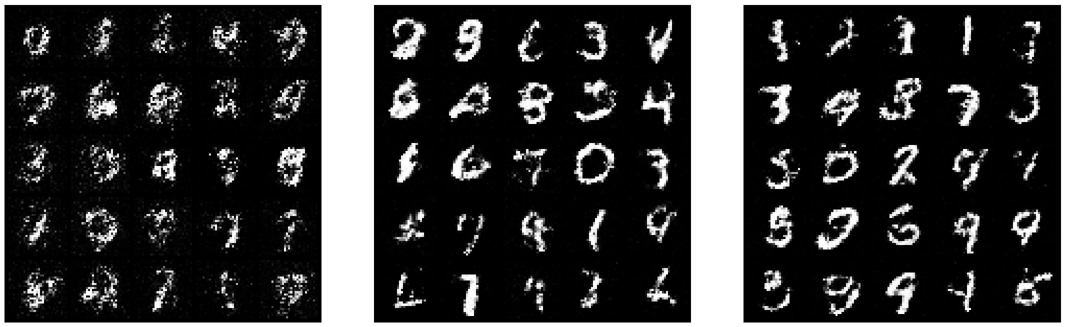

Figure 10: Plotted samples from my NF implementation for epoch 0 (left), epoch 20 (center) and epoch 40 (right).

Again we can see that the quality of the generated images improves over time.

4 Conclusion

Question 4.1

In this report we have designed and implemented three different generative models, namely VAEs, GANs and NFs. Different approaches underpin these three models, and each exhbits different properties. Iniially, VAEs and NFs belong to the explicit density family, using maximum likelihood, and GANs belong to the implicit density family of generative models. As we have seen, VAEs compress the data into a latent space with lower dimensionality, where GANs represent data with a latent distribution over noise vectors, were NFs do not preserve the dimensionality. This makes VAEs and GANs much faster to sample from, and to train because they deal with less dimensions. On the other hand, NFs are more stable to train than VAEs, and GANs. GANs are notoriously hard to train, however, I did not encounter any problems. Additionally, I found that GANs generate the best images, followed by VAEs and then NFs. The order of the last two actually depends on the dimensionality of the latent space for VAEs, as decreasing the dimensionality improves the quality of the generated images. Finally, these properties give each approach different capabilities, giving each model different pros and cons, therefore we can select the model depending on the nature of the problem.

References

Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., and Abbeel, P. (2016). Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 2172–2180. Curran Associates, Inc.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). Density estimation using real nvp.

Ho, J., Chen, X., Srinivas, A., Duan, Y., and Abbeel, P. (2019). Flow++: Improving flow-based generative models with variational dequantization and architecture design. *arXiv preprint arXiv:1902.00275*.

Rezende, D. J. and Mohamed, S. (2015). Variational inference with normalizing flows. *arXiv preprint arXiv:1505.05770*.

Seita, D. (2017). Understanding generative adversarial networks.