Deep Learning
Assignment 3

Nil Stolt Ansó (12371149)

1 Variational Auto Encoders

Question 1.1

1. A standard autoencoder bases its main function on minimizing the reconstruction error of the output based on the input. In the other hand a variational autoencoder's main function attempts to maximimze the probability that the output is representative of the underlying distribution of inputs.
2. A standard autoencoder is not a generative model as it does not assume input X to have a probability distribution and it does not attempt to map X into a probability function latent space z. With a standard AE's decoder, we can only reconstruct an image given the image's latent variables, but the model offers no information about the probability distribution of those latent variables.
3. Yes. A VAE can be used in place of a standard AE. In some sense, a VAE can be seen a version of AE's where not only the encoding of the latent variables is learnt, but also the underlying probability distribution of the data.
4. As opposed to standard AEs, a VAE explicitly models the probability distribution of the latent variables produced by the encoder. By taking the Bayesian approach, we can over time learn this distribution. Once we do this, we can sample from the distribution in order to generate data that might not have necessarily existed in the training set.

Question 1.2

In order to sample from provided the model, one has to take a hierarchical approach to sampling based on topological order. We begin by sampling variables that have no parents, in other words, variables that are not dependent on other variables. In the case of the given model, that would be $p\left(z_n\right) = \mathcal{N}\left(0, I_D\right)$ since it isn't dependant on anything. We would then proceed by transforming the sample through $f_\theta\left(z_n\right)$. Lastly, we would sample from the Bernoulli distribution (eq. 1) with our parent variables obtained so far (ie. $f_\theta\left(z_n\right)$) [1].

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

Question 1.3

Given a variable $\boldsymbol{z}$ from one distribution (in this case $\mathcal{N}\left(0, \boldsymbol{I}_D\right)$), we can create another random variable $X = g(z)$ with a completely different distribution, since the given Bernoulli equation 1 assumes we have a function $f_\theta\left(\boldsymbol{z}_n\right)_m$ that maps the distribution of z into the Bernoulli distribution. Assuming this function is complex enough to approximate the transformation, which in the case of a large enough neural network it is, we can safely assume any mapping between distributions is possible to be learn from data [2].

Question 1.4

a)
Say $p\left(\boldsymbol{x}_n|\boldsymbol{z}_n\right)=f_\theta\left(z_n\right)$, then:

$$\begin{aligned}\log p(\mathcal{D}) &= \sum_{n=1}^{N}\log p\left(\boldsymbol{x}_n\right)\\ &= \sum_{n=1}^{N}\log\int p\left(\boldsymbol{x}_n|\boldsymbol{z}_n\right)p\left(\boldsymbol{z}_n\right)dz_n\\ &= \sum_{n=1}^{N}\log\mathbb{E}_{p(z_n)}\left[p\left(\boldsymbol{x}_n|\boldsymbol{z}_n\right)\right]\\ &= \sum_{n=1}^{N}\log\left(\frac{1}{J}\sum_{j=1}^{J}f_\theta\left(\boldsymbol{z}_j\right)\right)\\ &= \sum_{n=1}^{N}\log\left(\frac{1}{J}\sum_{j=1}^{J}\boldsymbol{x}_j\right)\end{aligned}$$

In other words, by sampling from out posterior enough, we can get an estimate of the shape of our posterior distribution, and thus its integral.

b)
The inefficiency arises from having to sample. A good estimate requires a very large sample size, which can become computationally expensive. Estimating by sampling will always introduce some variance in the resulting integral value, which in turn will cause the model to require many more update states to converge because individual steps are in average less effective at bringing the model to an optimum.

Furthermore, Monte-Carlo integration suffers from the curse of dimensionality. Achieving a certain confidence by sampling from an n-dimensional space is less efficient than doing so in a $(n-1)$-dimensional space since the volume of n dimensional space from which to sample from grow exponentially larger with the number of dimensions. In short, it becomes exponentially less efficient to approximate the integral using sampling the larger the dimensionality of $\boldsymbol{z}_n$ is.

Question 1.5

The two following sections are based on the closed form solution derivations given here [3].
a)
The closed form derivation of the KL divergence of univariate gaussians:

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

$$D_{\mathrm{KL}}(q\|\mathcal{N}(0,1))=\log\frac{1}{\sigma_q}+\frac{\sigma_q^2+\mu_q^2}{2}-\frac{1}{2}$$

The gaussian distribution for $q=\mathcal{N}\left(\mu_q,\sigma_q^2\right)$ that would yield the lowest KL divergence is when both distributions p and q overlap, and thus the KL divergence would be 0.

$$\log\frac{1}{\sigma_q}+\frac{\sigma_q^2+\mu_q^2}{2}-\frac{1}{2}=0$$

$$\log\frac{1}{\sigma_q}+\frac{\sigma_q^2+\mu_q^2}{2}=\frac{1}{2}$$

If σ_q is equal to 1, then $\log \frac{1}{\sigma_q} = 0$ and $\frac{1+\mu_q^2}{2} = \frac{1}{2}$, thus μ_q must be equal to 0.
The KL divergence $D_{\mathrm{KL}}(q \| \mathcal{N}(0,1))$ is minimal when $q = \mathcal{N}(0,1)$

If we want to try to find a distribution $q = \mathcal{N}\left(\mu_q, \sigma_q^2\right)$ where the KL divergence is very large, we want to maximize equation 2.

The term $\log \frac{1}{\sigma_q}$ becomes maximal when σ_q tends to zero and the term $\frac{\sigma_q^2+\mu_q^2}{2}$ becomes maximal when μ_q tends to infinity or negative infinity. In other words, the KL divergence becomes very large the more peaked and further away the distribution $q = \mathcal{N}\left(\mu_q, \sigma_q^2\right)$ is from $p = \mathcal{N}(0,1)$

b)
The multivariate closed-form solution to $D_{\mathrm{KL}}(q \| p)$, as given here [4], is:

$$\begin{aligned}
D_{\mathrm{KL}}(q \| p) &= \int \left[\frac{1}{2} \log \frac{|\Sigma_p|}{|\Sigma_q|} - \frac{1}{2}\left(x-\mu_q\right)^T \Sigma_q^{-1}\left(x-\mu_q\right) + \frac{1}{2}\left(x-\mu_p\right)^T \Sigma_p^{-1}\left(x-\mu_p\right)\right] \times p(x) dx \\
&= \frac{1}{2} \log \frac{|\Sigma_p|}{|\Sigma_q|} - \frac{1}{2} \operatorname{tr}\left\{E\left[\left(x-\mu_q\right)\left(x-\mu_q\right)^T\right] \Sigma_p^{-1}\right\} + \frac{1}{2} E\left[\left(x-\mu_p\right)^T \Sigma_p^{-1}\left(x-\mu_p\right)\right] \\
&= \frac{1}{2} \log \frac{|\Sigma_p|}{|\Sigma_q|} - \frac{1}{2} \operatorname{tr}\left\{I_d\right\} + \frac{1}{2}\left(\mu_q-\mu_p\right)^T \Sigma_p^{-1}\left(\mu_q-\mu_p\right) + \frac{1}{2} \operatorname{tr}\left\{\Sigma_p^{-1} \Sigma_q\right\} \\
&= \frac{1}{2}\left[\log \frac{|\Sigma_p|}{|\Sigma_q|} - d + \operatorname{tr}\left\{\Sigma_p^{-1} \Sigma_q\right\} + \left(\mu_p-\mu_q\right)^T \Sigma_p^{-1}\left(\mu_p-\mu_q\right)\right] \qquad (3)
\end{aligned}$$

Question 1.6

We are trying to optimize our model by maximizing the log probability of the data $\boldsymbol{x}_n$. The KL divergence can never be negative, it is always positive or zero. We can remove the KL term on the left to get an inequality that expresses that the log probability is always going to be equal or larger than the term on the right depending on how much our two distributiions overlap:

$$\begin{aligned}
\log p\left(\boldsymbol{x}_n\right) - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right) &= \mathbb{E}_{q\left(z|\boldsymbol{x}_n\right)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p(Z)\right) \\
\log p\left(\boldsymbol{x}_n\right) &\geq \mathbb{E}_{q\left(z|\boldsymbol{x}_n\right)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p(Z)\right)
\end{aligned}$$

The term on the right is thus the absolute lower bound that $\log p\left(\boldsymbol{x}_n\right)$ can ever take (i.e. when $D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right) = 0$

Question 1.7

The log probability of the data $\log p\left(\boldsymbol{x}_n\right)$ we cannot directly manipulate and the functions $q\left(Z|\boldsymbol{x}_n\right)$ and $p\left(Z|\boldsymbol{x}_n\right)$ in $D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right)$ on the left we cannot express explicitly and thus we cannot make one more similar to the other to make the KL divergence tend to zero. The only variable we can interact with is the lower bound on the right (more specifically $\mathbb{E}_{q\left(z|\boldsymbol{x}_n\right)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right]$), which we can incrementally increase in other to optimize the left-hand-side (i.e. maximize $\log p\left(\boldsymbol{x}_n\right)$ and minimize $D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right)$).

Question 1.8

The two things that can happen is either the probability of the data $\log p\left(\boldsymbol{x}_n\right)$ is maximized, or the KL divergence between the two distributions (encoder and decoder) $D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right) \| p\left(Z|\boldsymbol{x}_n\right)\right)$ is minimized.

Question 1.9

$\mathcal{L}_n^{\text{recon}} = -\mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta\left(x_n|Z\right)\right]$ is termed *reconstruction loss* because we aim to minimize the expected reconstruction error of our decoder p given a latent space sample of our encoder q. In other words, by minimizing the reconstruction error, we are maximizing the log-probability of our decoder reconstructed data from the distribution that our decoder encodes into.

$\mathcal{L}_n^{\text{reg}} = D_{\text{KL}}\left(q_\phi\left(Z|x_n\right)\|p_\theta(Z)\right)$ is termed the *regularization loss* because it attempts to penalize the flexibility of the model (so as to reduce overfitting). It works similarly to L1 regularization in the sense that it penalizes large parameters deviating away from 0, except that rather than using a norm distance metric, it uses a distance metric (KL divergence in this case) from a standard distribution. In this case the standard distribution is $\mathcal{N}(0,1)$ since that is the form we assume $p_\theta(Z)$ takes. In other words, the further away from $\mathcal{N}(0,1)$ our function q_θ maps our input $\boldsymbol{x}_n$ into Z, the larger the regularization loss is going to be.

Question 1.10

1. First we sample input $\boldsymbol{x}_n$ from our dataset
2. Pass it through our encoder to obtain our $\boldsymbol{\mu}_n$ and $\boldsymbol{\sigma}_n$.
3. Obtain our Z sample by using $\boldsymbol{\mu}_n$ and $\boldsymbol{\sigma}_n$ on $\mathcal{N}(0,1)$
4. Use decoder to reconstruct image $\boldsymbol{x}_n$

Given that we chose to model our reconstruction probability as a Bernoulli distribution, given equation 1, our *reconstruction loss* will be:

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

Using our closed form solution for the KL divergence of two multivariate gaussian derived in 3, our *regularization loss* will be:

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

Question 1.11

a)
If we want to train the encoder at all, we need to take into account the effect that the encoder's parameters ϕ have on the loss. Otherwise, we do not know how to update those parameters.

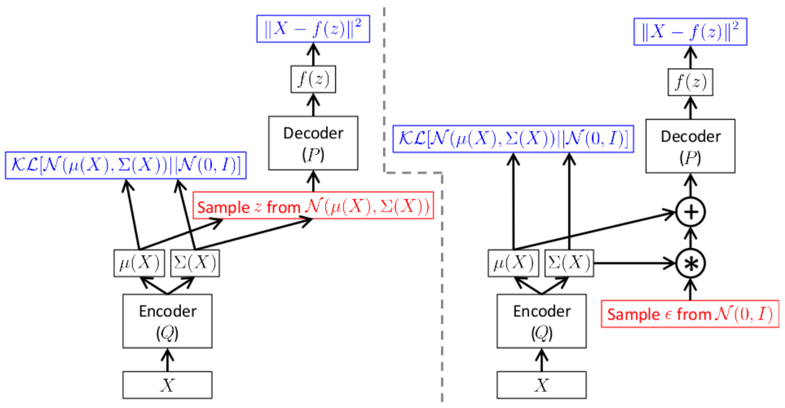
$\|X - f(z)\|^2$
$f(z)$
Decoder (P)
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X))||\mathcal{N}(0, I)]$
Sample z from $\mathcal{N}(\mu(X), \Sigma(X))$
$\mu(X)$
$\Sigma(X)$
Encoder (Q)
X
$\|X - f(z)\|^2$
$f(z)$
Decoder (P)
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X))||\mathcal{N}(0, I)]$
$+$
$*$
$\mu(X)$
$\Sigma(X)$
Sample ϵ from $\mathcal{N}(0, I)$
Encoder (Q)
X

Figure 1: A training-time variational autoencoder implemented as a feed- forward neural network, where $P(X|z)$ is Gaussian. Left is without the "reparameterization trick", and right is with it. Red shows sampling opera- tions that are non-differentiable. Blue shows loss layers. The feedforward behavior of these networks is identical, but backpropagation can be applied only to the right network.

b)
Since we are in some sense sampling from the latent probability distribution every time we encode a data point, there is a degree of stochasticity which doesn't allow us to compute a gradient backwards into the encoder. This is because the act of sampling from a distribution is a non-continuous operation which has no gradient.

c)
The *reparameterization trick* allows us to move the stochastic sampling into an input layer (see right model of Figure 1). Separating the act of sampling from the computation of $\mu(X)$ and $\Sigma(X)$ allows the gradient to propagate back into the encoder through those channels. During the forward pass the noisy sample from $\mathcal{N}(0, 1)$ is the translated and re-scaled with the encoder output by doing $z = \mu(X) + \Sigma^{1/2}(X) * \epsilon$.

Question 1.12

The implementation consists of an encoder and a decoder network. The encoder network has 2 heads which share the same hidden layer. One head predicts the means of the latent distribution, while the other predicts the log standard deviations (SD) of the latent distributions. The prediction of the log SD is done for numerical stability. When calculating the sample and the regularization loss, the log SD is exponentiatied to obtain the plain SD. The decoder takes in a 20-dimensional sample and outputs an image. The output layer passes the activation through a sigmoid function. This is to constrict the value from 0 to 1, as we aim to model the output as a Bernoulli distribution.

Question 1.13

The training and validation ELBO value seem to decrease rapidly for the first 10 or so epochs. From the on, it plateaus at a value around 80.0. The final validation ELBO was 79.458.

Question 1.14

The reconstructions of our latent space samples before any training, as seen in Figure 3, it is a bunch on noise. As seen in Figure 4, after a single epoch of training, the samples begin to be reconstructed into digit-resembling structures despite still being rather noisy. The largest

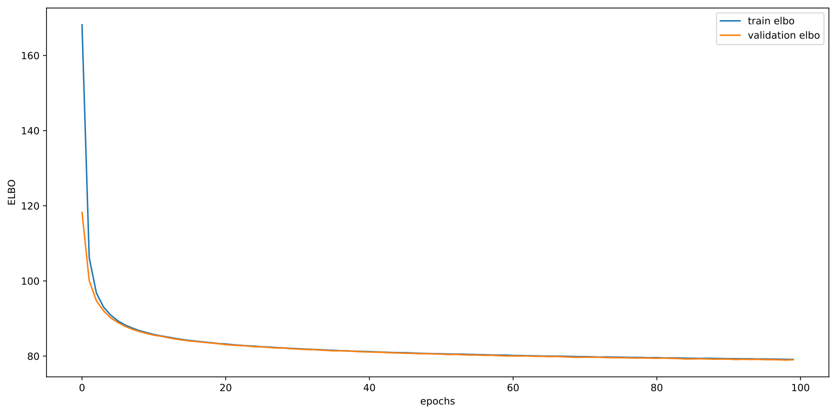

Figure 2: Training and validation ELBO values for a training period of 100 epochs.

takeaway from that image is that the decoder has learnt that the outskirts of the image should be black, as it has began to capture the information that the digits happen mainly in the center.
After 10 epochs, as seen in Figure 5, the strides of the digits begin to be sharper as most of the noise has began to disappear. Most of the digits begin to have a close resemblance to one of the 10 target digits, while others still have an appearance of a mixture of digits.
Finally after 100 epochs (Figure 6), there has been a slight improvement over the 10 epoch version (Figure 5), but there are still plenty of samples that are not reconstructed correctly, as they still maintain strucutures from multiple digits mixed together.

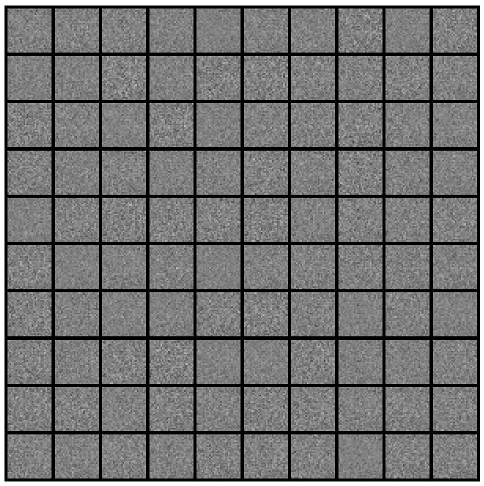

Figure 3: 100 reconstructed samples from our latent space before any training step was performed.

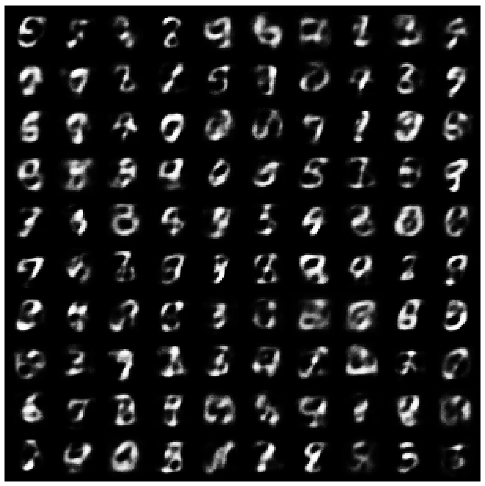

Figure 4: 100 reconstructed samples from our latent space after a single epoch of training.

Question 1.15

Setting the number of dimensions of the latent space to 2, allows us to explore the latent space in a single image, as seen in Figure 7. The grid points from which each generation was taken was created using the scipy package's *norm.ppf* function.

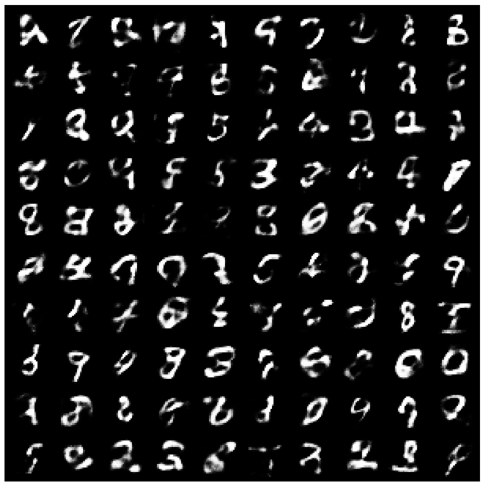
Figure 5: 100 reconstructed samples from our latent space after 10 epochs of training.

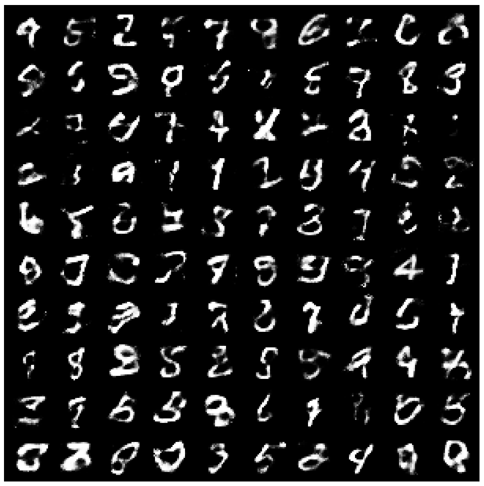
Figure 6: 100 reconstructed samples from our latent space after 100 epochs of training.

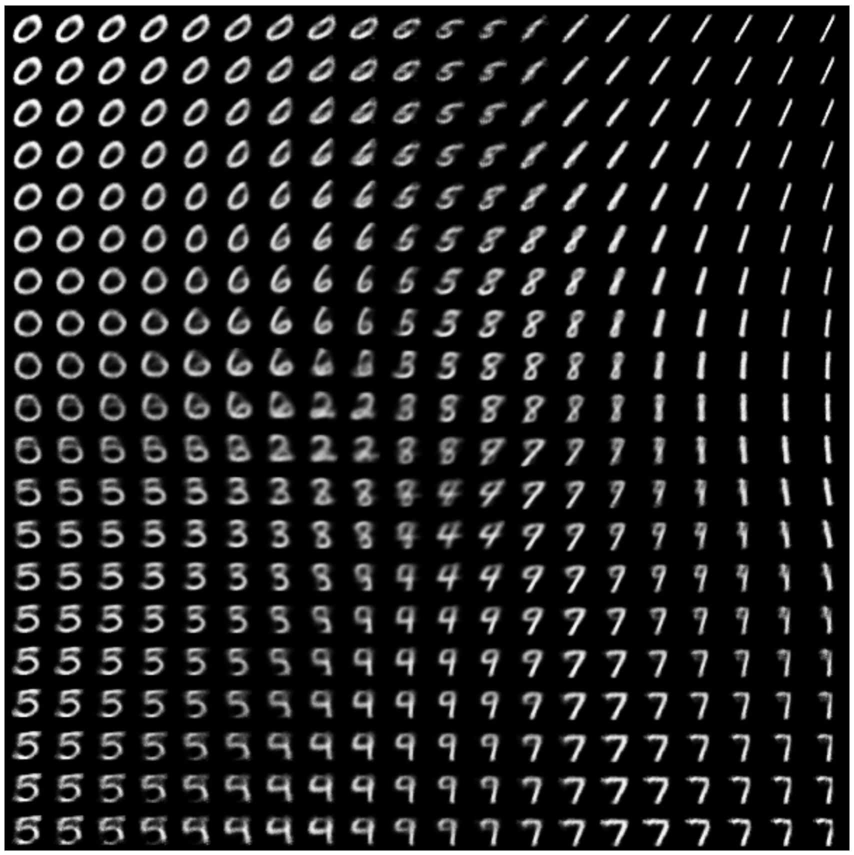
Figure 7: Data manifold for a 2-dimensional latent space VAE. The x and y directions represent values of our latent space ranging from negative to positive along the 2 dimensions of our latent distribution.

2 Generative Adversarial Networks

Question 2.1

The generator is a function that takes in a latent space sample $\boldsymbol{z}$ and outputs a generated image.
The discriminator is a function that takes in an image and outputs a single value representing the probability that the image originated from the real data set.

Question 2.2

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

The term $\mathbb{E}_{p \text{ data } (x)}[\log D(X)]$ is responsible for improving the prediction by the discriminator when the input image X is real. The generator is not responsible for anything about this term as seen by how only the discriminator $D(X)$ is present. If the discriminator make a good probability prediction when the input image is real, the loss generated by this term will be $\log(1) = 0$. In the other hand, the log of a probability prediction close to 0, will result in a very negative term.

The term $\mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))]$ is responsible for the learning of both the generator and discriminator when judging an image that was generated. The discriminator attempts to maximize this term, as the prediction $D(G(Z))$ should be ideally low thus tending towards $log(1 - 0) = 0$, while if the discriminator makes a bad prediction (close to 1.0) the term will be very negative. The generator aims to minimize this term, as the generator aims to have its output be classified as real. The generator thus aims to make the discriminators prediction as high as possible.

Question 2.3

The resulting value for $V(D, G)$ after convergence tends to $\log(0.5)$. The reason behind this is that as the generator converges to a minimum by having learned to generate indistinguishable images from the ones in the real image data set, the discriminator will practically be guessing at random if the image is real or not, i.e. $D(X) = 0.5$. Either of the two terms (depending on if a real image or generated is fed) will thus be $\log(0.5)$.

Question 2.4

Generally, it is easier for the discriminator to become proficient at telling real and generated images apart than it is for the generator to generate convincing images. Given that the generator starts off generating noise, it is easy for the discriminator to for example tell apart noisy features from common real image features. In such a case, since the critic is outputting a confident value that the image is fake, $\log(1 - D(G(Z)))$ will commonly be close to zero, thus not allowing much of the gradient to flow down to the generator.

A solution is to instead change the generators training objective to maximizing $\log(D(G(Z)))$. This way, even in the case the discriminator is very good at classifying generated images as fake, $D(G(Z))$ will be outputting values closer to 1, making the gradient $\frac{1}{D(G(Z))}$ which is larger the better the discriminator is in relation to the generator.

Question 2.5

The network structures used were those given in the template. As an output non-linearity for the generator, a tanh function was used. The reason behind this is that the data is normalized so as to have pixels be distributed in the range [-1, 1]. A output range of a tanh is thus a more natural fit to the data.
During the training procedure, rather than computing the log-likelihoods manually, Torch's Functional packages *binary_ cross_ entropy* function is used for simplicity. After sampling a

batch z of gaussian noise, the generator generates images and is trained using the alternate loss $\log(D(G(Z)))$ mentioned in section 2.4. Using the flag*retain_graph=True* during the *.backward()* call, allows us to not have to forward propagate the generator again before backpropagating through the discriminator.
The discriminator is then trained using a slightly modifed version of the GAN objective seen in 4. A technique known as *one-sided label smoothing* is used where we prevent the discriminator from becoming overconfident by making it's training objective be 0.9 rather than 1.0 [5]. This helps the generator receive at the very least a bit of gradient signal even when the discriminator is very good.

The loss obtained throughout training can be seen in Figure 8.

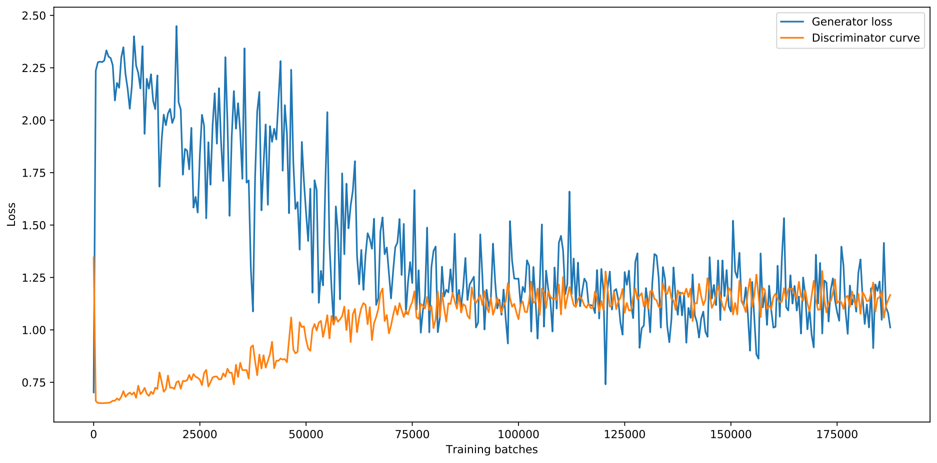

Figure 8: Discriminator (orange) and generator (blue) loss throughout training procedure. Loss is obtained from PyTorch's *binary_cross_entropy* function. Discriminator and generator seem to converge at around the same loss.

Question 2.6

As we would expect, Figure 9 shows how the generator outputs noise at the beginning of the training. After 10000 training batches (around 4.5% through the training), despite the noise, the generator begin to learn that the digits should be white in the center, and that the surrounding background should be black (Figure 10).
After 99000 training batches (approximately 50% through the training), we can see in Figure 11 that the generator begins to generate patterns reminiscent of digits. For the most part, they are noisy and some have structre belonging to different digits, but they are starting to be clear. It should also to be noted that the generator seem to be at a local optimum where it seems to have learnt a bias towards generating '3's to fool the discriminator.
After the entire training procedure of 187500 training batches, Figure 12 shows that the network is very good at generating clear digits without much noise in them. Only one sample (bottom-left) seems to be rather unclear.

Question 2.7

After doing some exploring through the latent space, decided to use the latent space point corresponding to a vector where each element is -0.9 and the latent space point where each element in the vector is 0.9. These points correspond to a '5' digit and a '9' digit respectively, as seen in Figure 13. Moving through latent space in equally spaced intervals, lead to the reel of pictures seen in Figure 13.

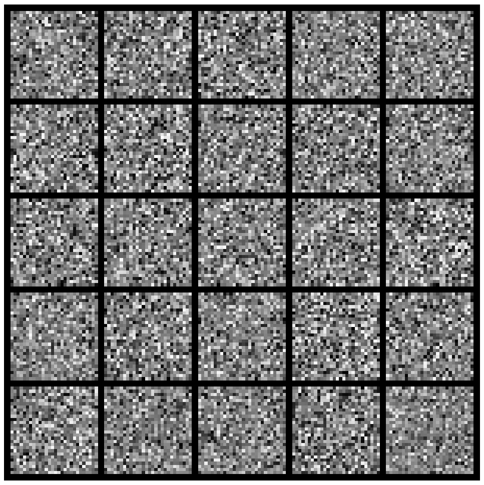
Figure 9: 25 generated digits after a single batch of training.

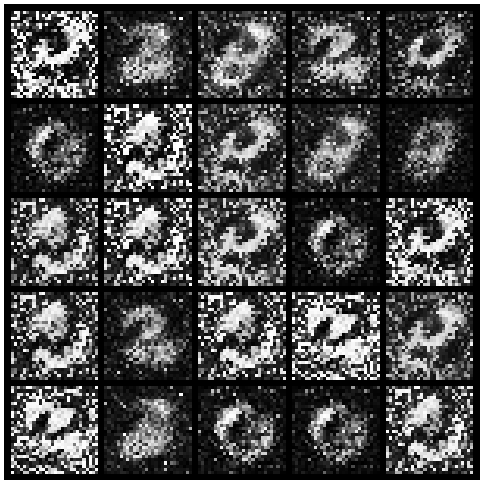
Figure 10: 25 generated digits after 10000 batches of training.

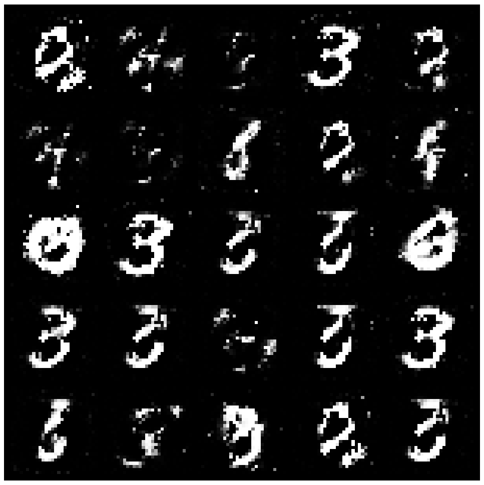
Figure 11: 25 generated digits after 99000 batches of training.

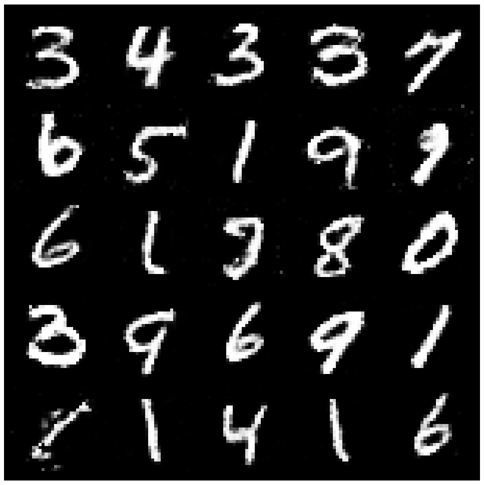
Figure 12: 25 generated digits after 187500 batches of training.

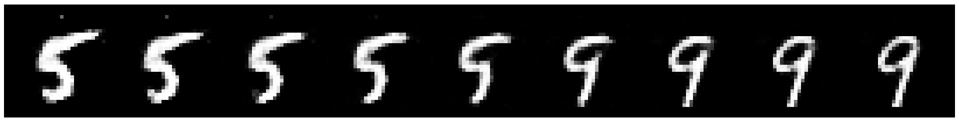
Figure 13: Interpolation through latent space between a '5' and a '9' in equally spaced intervals.

3 Normalizing flows

Question 3.1

For a mapping $f : \mathbb{R}^m \to \mathbb{R}^m$ of a multivariate random variable, the derivative $\frac{df}{dx}$ is expressed as the determinant of the Jacobian [6].

$$p(x) = p(z) \left| \det \left(\frac{\partial f}{\partial x^T} \right) \right|$$

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

Question 3.2

Every layer of the neural network h_l must be continuous and invertible. Using a non-linearity such a ReLU will not work. Furthermore, since we have to obtain the determinant of the Jacobian, the Jacobian must thus be a square matrix, meaning that the amount of input dimension and output dimensions to the layer must remain the same $f : \mathbb{R}^m \rightarrow \mathbb{R}^m$. Since we are aiming to use log-probabilities, the determinant must be positive, since the log of zero or a negative value is undefined.

Question 3.3

Computing the determinant of any arbitrarily sized square matrix has a computational complexity $O(N^3)$, which is not ideal for training a neural network. This is specially the case when applying Normalizing Flows to the area of computer vision, where we tend to require very large and deep networks, since each layer requires this operation.

Furthermore, the process of summing the determinant of every coupling layer is usually strictly serial since you have to compute the forward propagation of a coupling layer before you can obtain its determinant. Research is being done on ways to parallelize these operations, as well as finding types of layers that are cheap to invert and compute the gradient for.

Question 3.4

The cumulative density function, when discrete input with large gaps is fed to the model, will become sparse with occasional large spikes on the values where the input always occurs. For example, given that pixel values range from [0, 255], our model likely to spike at every integer value.

The paper 'Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design' [7] offers the following improvement: adding uniform noise to the input for dequantization in order to fill in the input range [0, 255] evenly so as to make the CDF smoother.

I can further imagine downscaling the input to a smaller range like for example [0,1) might make the input appear less numerically discrete to the network, since the input distribution is then more compact in relation to its weight initialization.

Finally another suggestion I can imagine having an impact is modelling the input as a wave through the use of Fourier synthesis. This technique has been applied to Graph Neural Networks to be able to model the value of several nodes as a continuous function that can be manipulated to allow for gradient descent.

Question 3.5

During training, we have an input a n-dimensional image $\boldsymbol{x}$. An image is typically flattened into an n-dimensional vector. The output will be single value corresponding to the log-probability of that image.

During inference, we sample a vector $\boldsymbol{z}$ from the latent distribution which serves as input. We use $\boldsymbol{z}$ to generate an image through the inverted flow through the coupling layers.

Question 3.6

During training, we first preprocess our data so as to, as mentioned in section 3.4, dequantize the data to approximate better a continuous variable. Then we initialize our network's weights for every layer. After this we begin training by sampling a batch and feeding it to the model. The data thus propagates through our model and we obtain a latent vector. We use the determinants of our coupling layers to obtain the log-probability of the input data. At the end of each step, we minimize the negative log-probability by propagating it through our layers.

During inference, we first sample a latent vector. Since our coupling layers are all invertable, we use the inverse of our coupling layers to propagate our latent vector in reverse of the flow so as to obtain a data point.

Question 3.7

See the attached *a3_nf_template.py* file. The neural network responsible for calculating the scaling and the translation is made up of a shared body of 2 linear layers with ReLU activation functions. Two separate heads use these shared layers, one that ouputs the log-scale dimensions and the other the translantion. In constrast to the Real NVP paper [8], when computing the sum of determinants across all layers, my implementation performs $\sum_j s\left(x_{1:d}\right)_j$ when computing a layer's determinant (as apposed to $\exp\left[\sum_j s\left(x_{1:d}\right)_j\right]$) due to my network predicting the log scale. Furtherore, the gradient at each step is clipped to a maximum of 5.0 in order to avoid gradient explosions.

Question 3.8

As seen in Figure 14, after one epoch the BPD value is around 2.1 for validation. Compared to the samples generated before any training epochs (Figure 15) which appear as noise, after 1 epoch (Figure 16) the background is being generated correctly as black and the silhouette of digits has began to form despite the patterns themselves still appearing noisy.

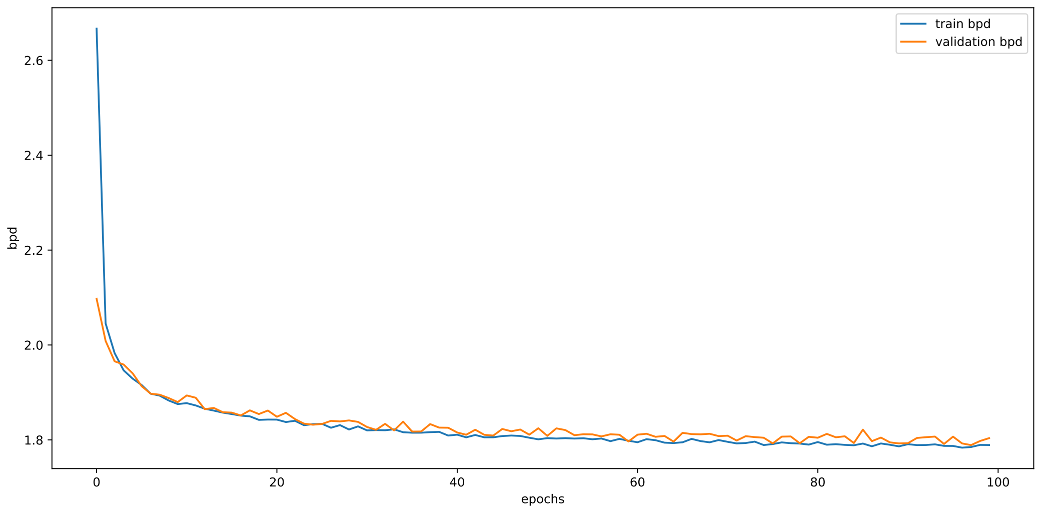

Figure 14: Training (blue) and validation(orange) bits per dimension (BDP) across 100 training steps.

After 10 epochs (Figure 17), the digits are much more well formed and less noisy. For the most part the form structures of only one digit, but in some cases some generated samples still share features of multiple classes. The BPD at this stage, as seen in Figure 14, has began to plateau at this point at around 1.85.

After 40 epochs, the BPD reaches a validation value of 1.826. Finally, after 100 epochs, the validation BPD is 1.804. The resulting samples after the model is trained on 100 epochs can be seen in Figure 18. The generated digits are still somewhat noisy and as for the structure, although the target class is recognizable in the majority of cases, it isn't as clear as one would hope.

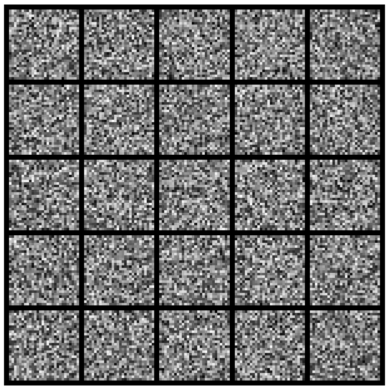
Figure 15: 25 reconstructed samples from our latent space before any training step was performed.

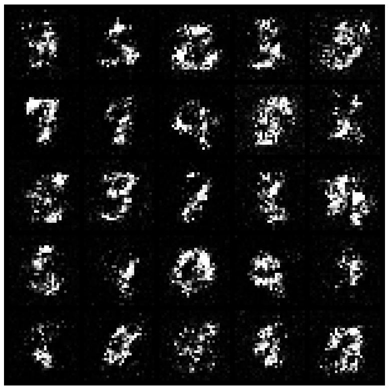
Figure 16: 25 reconstructed samples from our latent space after a single epoch of training.

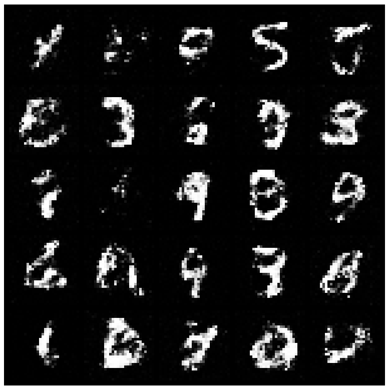
Figure 17: 25 reconstructed samples from our latent space after 10 epochs of training.

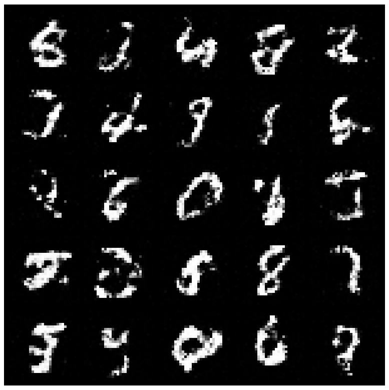
Figure 18: 25 reconstructed samples from our latent space after 100 epochs of training.

4 Conclusion

Between the VAEs, GANs, and NFs, the easiest to get good results from are GANs. The update function of a GAN is very quite simple and intuitive. The downsides of GANs are that training can be unstable due to phenoma such as node collapse or be slow due to small gradients when the performance of the networks is not in balance.
VAEs and NFs have the upside of allowing the explicit calculation of the generater sample's probability, but they do not come without their downsides, with one of the main ones being how mathematically complex the systems need to be in order to achieve comparable performance to a GAN. VAEs have to make the explicit assumption of the latent space's distribution, which is not a straightforward thing to do, as different data might require different latent space distributions. Furthermore, the latent space might suffer from 'holes', causing samples to fail at generation. Also, despite the use of the *reparametrization trick*, gradients might have large variance.
NFs solve some of VAE's issues with latent spaces by not having an explicit bottleneck, but this might mean that the number of parameters explodes for tasks involving large input. On

top of this, NFs are not as computationally efficient as the other two methods as they are not easily parallelizable.

References

[1] Sampling methods. `https://ermongroup.github.io/cs228-notes/inference/sampling/`. Accessed: 2018-05-14.

[2] Tutorial on variational autoencoders. `https://arxiv.org/abs/1606.05908/`. Accessed: 2018-05-14.

[3] Univariate kl divergence closed form derivation. `https://stats.stackexchange.com/questions/7440/kl-divergence-between-two-univariate-gaussians`. Accessed: 2018-05-14.

[4] Multivariate kl divergence closed form derivation. `https://stats.stackexchange.com/questions/60680/kl-divergence-between-two-multivariate-gaussians`. Accessed: 2018-05-14.

[5] Gan — ways to improve gan performance. `https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b`. Accessed: 2018-05-19.

[6] Normalizing flows tutorial, part 1: Distributions and determinants. `https://blog.evjang.com/2018/01/nf1.html`. Accessed: 2018-05-19.

[7] Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. Flow++: Improving flow-based generative models with variational dequantization and architecture design. *CoRR*, abs/1902.00275, 2019.

[8] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real NVP. *CoRR*, abs/1605.08803, 2016.