Deep Learning - Assignment 3: Deep Generative Models

Ioannis Gatopoulos 12141666

May 18, 2019

Assignment 3 of Deep Learning Course 2019
MSc Artificial Intelligence • University of Amsterdam

Abstract

In these project, we are going to study and implement three of the most famous and powerful generative models, namely Variational Auto Encoders (VAEs), Generative Adversarial Networks (GANs) and Generative Normalizing Flows (NFs). We are going to analyze both from theoretical and practical spectrum, as we are going to present their mathematical framework and results for practical implementations. Besides that they manage to generate samples like the input ones, that did not exist before, these models are really important because the manage to learn also the underling distribution of the real data. Like the famous and great physicist Richard Feynman said: *what I cannot create, I do not understand.*

Contents

1 Variational Auto Encoders

1.1 Latent Variable Models

Question 1.1

1. The standard autoencoder is an deterministic procedure that its objective is to minimize the reconstruction loss. They can be used for applications like denoising and unsupervised pre-training. One the other hand, Variational Autoencoders (VAEs) are used mainly as generative models, as you can pick a point from the latent space and generate a sample from it.

2. For generation, the fundamental problem with autoencoders is that, the latent space they map their inputs to and where their encoded vectors lie, may not be continuous, or allow easy interpolation. Thus, because the learned latent space has discontinues, there is a high chance that you sample a point that the decoder has not trained on, and generate a random sample. This is becomes clearer when looking on figure 1; as there is a formation of distinct clusters, the generation of the sample names `'?'`, will produce a random image, since the model has not map anything to that region. In general, for generative model, you do not want to replicate the same image that we put in. However, the latent spaces of VAEs are continuous, allowing easy random sampling and interpolation.

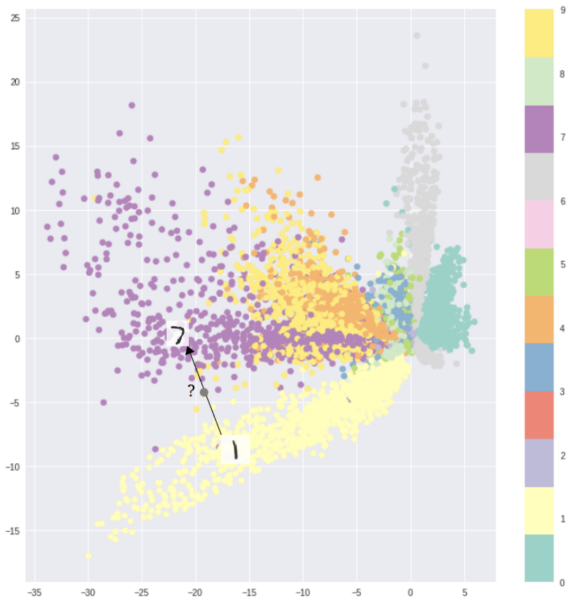

Figure 1: Visualization of the encodings of a trained autoencoder on the MNIST dataset, from a 2D latent space; it reveals the formation of distinct clusters

3. As standard autoencoders are considered to be a spacial case of VAEs, when KL-diverge is zero, they can be used in place of the former's.

4. As mentioned before, standard autoencoders are missing the continuity of the latent space, which allows a generative model to be able to generalize.

1.2 Decoder: The Generative Part of the VAE

Question 1.2

Assuming that the latent space has dimensionality D, the process of sampling from the decoder is

- We first sample a vector $\boldsymbol{z_n}$, one for every example that we want to generate, from the normal distribution $\mathcal{N}(\mathbf{0}, \boldsymbol{I}_D)$. In mathematical terms, $\boldsymbol{z} \sim \mathcal{N}(0,1)^{D_z}$
- We feed the above into the decoder f_θ, we obtain a $f_\theta(\boldsymbol{z}_n^m)$ for every dimension m of $\boldsymbol{z}_n$. which will output the mean for each sample.
- Finally, we can sample each binary pixel $\boldsymbol{z}_n^m$ from the *Bernoulli* distribution as $\tilde{\boldsymbol{x}}_n^m \sim \text{Bern}\left(f_\theta\left(\boldsymbol{z}_n^m\right)\right)$, $\forall m \in M$.

Question 1.3

Assuming that the prior follows just a standard norm, it will just force the latent variables $\boldsymbol{z}_n$ to be around zero and with some unique variants. As it is stated in [1]:

> *"The key is to notice that any distribution in d dimensions can be generated by taking a set of d variables that are normally distributed and mapping them through a sufficiently complicated function." ... ."Hence, provided powerful function approximators, we can simply learn a function which maps our independent, normally-distributed z values to whatever latent variables might be needed for the model, and then map those latent variables to X."*

Thus, in general, we do not need to worry about ensuring that the latent structure exists. If such latent structure helps the model accurately reproduce (i.e. maximize the likelihood of) the training set, then the network is powerful enough to learn mapping from a standard normal to that structure.

Question 1.4

a) The Monte Carlo estimation states that

$$\mathbb{E}_{p(x)} f(x) \approx \frac{1}{M} \sum_{s=1}^{M} f\left(x_s\right), \quad x_s \sim p(x)$$

Therefore, we can approximate the given expected value as

$$\log p\left(x_n\right) = \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p\left(\boldsymbol{x}_n|\boldsymbol{z}_n\right)\right] \approx \log \frac{1}{M} \sum_{m=1}^{M} p\left(\boldsymbol{x}_n|z_n^m\right), \quad z_n^m \sim p(\boldsymbol{z}_n)$$

b) Looking at 2, we can see that $p(x|z)$, the generated samples, lives in a very small space are relatively to the prior space. Sampling any point other than this area, will not produce a meaningfully sample related to our true samples, but just noise. Thus, referring to figure 2 again, most of the samples z fall outside of the $p(x|z)$ region, making MC highly inefficient. We can easily understand that as the dimensions of the latent space grows, this ratio of samples that fall in $p(x|z)$ to all others increases as well.

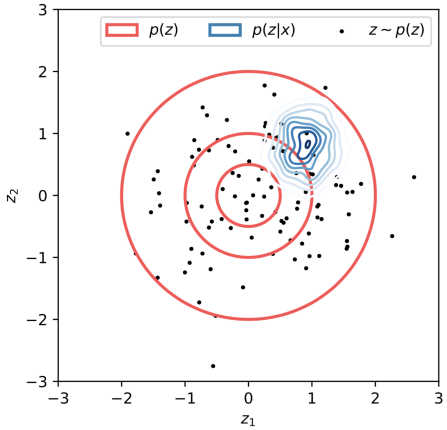

Figure 2: Plot of 2-dimensional latent space and contours of prior and posterior distributions.

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

Question 1.5

a) For the $\mathcal{KL}$-divergence it is known that $\mathcal{KL}(q\|p) \geq 0$, $\forall p, q$. Especially, if $p = q$, then $\mathcal{KL}(q\|p) = 0$. Therefore, for $q \sim \mathcal{N}(0, 1)$, we will have the lowest $\mathcal{KL}$ difference there is, 0.

Taking a look at the closed-form solution of $\mathcal{KL}$-divergence between two normal distributions (0.1), since $\sigma_p = 1$ and $\mu_p = 0$, by making σ_q very small, $\sigma_q \to 0$, the $-\log \sigma_q$ and the will be very large, resulting up to infinitely large value of their $\mathcal{KL}$ difference. Thus, $q \sim \mathcal{N}(0, 0.0001)$, will cause a very large $\mathcal{KL}$-divergence value.

b) The variational lower bound (the objective to be maximized) contains a KL term that can often be integrated analytically. The (closed-form) formula for $D_{\mathrm{KL}}(q\|p)$ is

$$D_{KL}(q\|p) = \int q \log \frac{q}{p} \overset{*}{=} \log \frac{\sigma_p}{\sigma_q} + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} - \frac{1}{2} \tag{0.1}$$

and for $p \sim \mathcal{N}(0, 1)$, the above becomes

$$D_{KL}(q\|p) = \log \frac{1}{\sigma_q} + \frac{\sigma_q^2 + \mu_q^2}{2} - \frac{1}{2} \tag{0.2}$$

Finally, as noted in the appendix of [1], in case of multivariate Gaussian distribution, the (0.2) can be written in the following form

$$D_{KL}\left(q_\phi(\mathbf{z})\|p_{\boldsymbol{\theta}}(\mathbf{z})\right) = -\frac{1}{2}\sum_{j=1}^{D}\left(1 + \log\left((\sigma_j)^2\right) - (\mu_j)^2 - (\sigma_j)^2\right) \tag{0.3}$$

where D is the dimensions of the latent space.

Question 1.6

From the log-probability of of the data we have

$$\begin{aligned}
\log p(\boldsymbol{X}) \overset{iid}{=} \log \prod_{n=1}^{N} p(\boldsymbol{x}_n) = \sum_{n=1}^{N} \log p(\boldsymbol{x}_n) &= \sum_{n=1}^{N} \log \int p(\boldsymbol{x}_n, Z) dZ \\
&= \sum_{n=1}^{N} \log \int \frac{q(Z|\boldsymbol{x}_n)}{q(Z|\boldsymbol{x}_n)} p(\boldsymbol{x}_n, Z) dZ \\
&= \sum_{n=1}^{N} \log \mathbb{E}_{q(Z|\boldsymbol{x}_n)} \frac{p(\boldsymbol{x}_n, Z)}{q(Z|\boldsymbol{x}_n)}
\end{aligned} \tag{0.4}$$

Since the function $\log(\cdot)$ is a **concave** function, it is known that

$$f\left(\mathbb{E}_{p(t)} t\right) \geq \mathbb{E}_{p(t)} f(t)$$

*The analytically solution is provided to the appendix.

Thus, going back to the (0.4), we will have

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

Thus, we proved that

$$\log p(\boldsymbol{X}) \geq \mathcal{L}(\theta, \phi)$$

making $\mathcal{L}(\theta, \phi)$ a *lower (variational) bound* of the log-probability of of the data $\log p(\boldsymbol{X})$.

Note: Another way to justify is by looking to the the expression for the KL-divergence between our proposal $q\left(z_n|x_n\right)$ and our posterior $p\left(z_n|x_n\right)$.

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

Because, as we mention already, $D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right)\|p\left(Z|\boldsymbol{x}_n\right)\right) \geq 0$, the above can be written as

$$\log p\left(\boldsymbol{x}_n\right) \geq \mathbb{E}_{q(z|\boldsymbol{x}_n)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right)\|p(Z)\right) \iff \log p\left(\boldsymbol{x}_n\right) \geq \mathcal{L}(\theta, \phi)$$

Question 1.7

In order to optimize the log-probability directly, we must calculate the expression $D_{\mathrm{KL}}\left(q\left(Z|x_n\right)\|p\left(Z|x_n\right)\right)$, which is not possible, since the true posterior $p(Z|x_n)$ is unknown to us. Moreover the analytical expression of $p(Z|x_n)$ contains integral with neural networks in it, which, in general, is intractable.

Note: Looking at the log-probability of of the data, we understand that to optimize it, we have to maximize the expression

$$\max_{\theta} \log p(\boldsymbol{X}|\theta) = \max_{\theta} \sum_{n=1}^{N} \log \int p(\boldsymbol{x}_n, Z|\theta) dZ = \max_{\theta} \sum_{n=1}^{N} \log \int p(\boldsymbol{x}_n|Z, \theta) p(Z) dZ$$

In general, this integral is intractable, because now, the $p(\boldsymbol{x}_n|Z, \theta)$ contains neural networks in it. So we can not compute even one point. We have to note that this expression can be computed if the distributions are **conjugated**, which is the case on **pPCA** $\Big(p\left(Z\right) = \mathcal{N}(0, I)$ and $p(\boldsymbol{x}_n|Z, \theta) = \mathcal{N}\left(Wz_i + b, \Sigma\right)\Big)$. Additionally, since we have a latent variable model, our intuition is to use the **EM** algorithm. Sadly, we also can not use it, since at the E-step is intractable, as we have to compute the posterior distribution $p(Z|X, \theta)$, which contains again neural networks inside it and it is too difficult to find it analytically. Therefore, the solution is to use a approximation of the **Variational EM** algorithm.

Question 1.8

Re-witting the expression for the KL-divergence between our proposal $q\left(z_n|x_n\right)$ and our posterior $p\left(z_n|x_n\right)$, we get

$$\begin{aligned}
&\log p\left(\boldsymbol{x}_n\right) - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right)\|p\left(Z|\boldsymbol{x}_n\right)\right) = \mathbb{E}_{q(z|\boldsymbol{x}_n)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right)\|p(Z)\right) \\
\iff &\log p\left(\boldsymbol{x}_n\right) = \mathbb{E}_{q(z|\boldsymbol{x}_n)}\left[\log p\left(\boldsymbol{x}_n|Z\right)\right] - D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right)\|p(Z)\right) + D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right)\|p\left(Z|\boldsymbol{x}_n\right)\right) \\
\iff &\log p\left(\boldsymbol{x}_n\right) = \mathcal{L}(\theta, \phi) + D_{\mathrm{KL}}\left(q\left(Z|\boldsymbol{x}_n\right)\|p\left(Z|\boldsymbol{x}_n\right)\right)
\end{aligned}$$

There are two possible scenarios:

- If the lower-bound is equal to the log marginal likelihood, then the KL-distance between the true posterior and its approximation is zero. Thus, the q distribution is set equal to the posterior distribution.
- If the lower-bound is not equal to the log marginal likelihood, the difference is covered by the KL-distance between the true posterior and q. Thus, distribution q under-approximates the true posterior.

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

Question 1.9

The loss as the mean negative lower bound is defined as

$$\mathcal{L}(\theta,\phi) = -\frac{1}{N}\sum_{n=1}^{N} \mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta\left(x_n|Z\right)\right] - D_{\mathrm{KL}}\left(q_\phi\left(Z|x_n\right)\|p_\theta(Z)\right) = \frac{1}{N}\sum_{n=1}^{N}\left(\mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}\right) \tag{0.5}$$

Reconstruction loss term In order to make clear the reasons why and how, lets consider the case that $p_\theta\left(\boldsymbol{x}_n|\boldsymbol{z}_n\right)$ follows a Gaussian, thus $p(\boldsymbol{x}_n|\boldsymbol{z}_n,\theta) \sim \mathcal{N}(\mu(\boldsymbol{z}_n,\theta),\Sigma(\boldsymbol{z}_n,\theta))$. Additionally, lets suppose that the variance is constant, for example equal to 1. Now, if we take the logarithm of this Gaussian, we result into $-\left\|\boldsymbol{x}_n - \mu\left(\boldsymbol{z}_n,\theta\right)\right\|^2 + \text{const}$. Thus, in order to maximize it (maximize the lower bound), we want $\mu\left(\boldsymbol{z}_n,\theta\right)$ as close as possible to the $\boldsymbol{x}_n$. So this is actually a reconstructive loss, where the neural net tries to bring them close, which is actually tho objective of a autoenconder. We see that the expectation is taken with respect to the encoder's distribution over the representations and, therefore, this term encourages the decoder to learn to reconstruct the data. If the decoder's output does not reconstruct the data well, statistically we say that the decoder parameterizes a likelihood distribution that does not place much probability mass on the true data. In general, for the $\boldsymbol{z}_n$ that are likely to cause $\boldsymbol{x}_n$, according to our encoder, the reconstruction loss to be low.

Regularization term This term pushes the encoder to be non-deterministic, but stochastic and is maybe the most important part of the VAEs. When we maximize the $-D_{KL}$, we actually try to minimize the D_{KL}, thus we try to push the encoder as close as to the prior. Recall that the KL, because the prior is just a standard normal distribution, becomes infinitely big as the variance of the encoder approximates zero ($\mathcal{KL}(q(z|x)\|p(z)) = \mathcal{KL}(\mathcal{N}(0,\approx 0)\|\mathcal{N}(0,1)) \approx \inf$). Thus, the loss will be infinitely big as well, and finally explodes. Therefore, the model instead of choosing to reduce completely the noise in itself, it will allow some.

Question 1.10

Given $\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$, where $\boldsymbol{x}_n \in \{0,1\}^M$, and D latent's space dimension, the procedure of calculating the ELBO loss would be:

- $\mathcal{L} = \frac{1}{N}\sum_{n=1}^{N}\left(\mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}\right)$
  - Calculate $\mathcal{L}_n^{\text{recon}}$, and since we have a Bernoulli MLP as decoder, the $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$ would be a multivariate Bernoulli

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

– Calculate $\mathcal{L}_n^{\text{reg}}$ using equation (0.3):

$$\begin{aligned}\mathcal{L}_n^{reg} &:= D_{KL}\left(q_\phi\left(\mathbf{z}_n|\mathbf{x}_n\right)\right)\|p_\theta\left(\mathbf{z}_n\right)) \\ &= -\frac{1}{2}\sum_{d=1}^{D}\left[\log\left(\left(\Sigma_\phi^{(d)}\left(\mathbf{x}_n\right)\right)^2\right) - \left(\mu_\phi^{(d)}\left(\mathbf{x}_n\right)\right)^2 - \left(\Sigma_\phi^{(d)}\left(\mathbf{x}_n\right)\right)^2 + 1\right]\end{aligned}$$

1.5 The Reparametrization Trick

Question 1.11

a) Since now, we have fully defined our model but no how to train it. Since now neural networks are involved, we want to use backpropagation and perform stochastic gradient descent on the loss function with respect to both parameters ϕ and θ, as they are the parameters that define the model and the loss. Both of these parameters are responsible and construct the family of lower bounds that we use to approximate the true posterior, and thus, to approximate it we will need to optimize both of them.

b) First of all, looking at the ELBO (0.5), the loss that we want to minimize, the term $\mathcal{L}_n^{\text{reg}}$ will not be a problem, since we can derive an analytical expression of it. The problem lies on the first term, and in particular, the derivation with respect to ϕ

$$\nabla_\phi \sum_{n=1}^{N} \mathbb{E}_{q_\phi(z|x_n)}\left[\log p_\theta\left(x_n|z_n\right)\right]$$

Taking this derivation step by step, we result to

$$\begin{aligned}\nabla_\phi \sum_{n=1}^{N} \mathbb{E}_{q_\phi(z_n|x_n)}\left[\log p_\theta\left(x_n|z_n\right)\right] &= \nabla_\phi \sum_{n=1}^{N} \int q_\phi\left(z_n|x_n\right)\log p_\theta\left(x_n|z_n\right)\, dz_n \\ &= \sum_{n=1}^{N} \nabla_\phi \int q_\phi\left(z_n|x_n\right)\log p_\theta\left(x_n|z_n\right)\, dz_n \\ &= \sum_{n=1}^{N} \int \nabla_\phi\, q_\phi\left(z_n|x_n\right)\log p_\theta\left(x_n|z_n\right)\, dz_n \\ &= \sum_{n=1}^{N} \int \frac{q_\phi\left(z_n|x_n\right)}{q_\phi\left(z_n|x_n\right)} \nabla_\phi\, q_\phi\left(z_n|x_n\right)\log p_\theta\left(x_n|z_n\right)\, dz_n \\ &\overset{\ddagger}{=} \sum_{n=1}^{N} \int q_\phi\left(z_n|x_n\right)\nabla_\phi\, \log q_\phi\left(z_n|x_n\right)\log p_\theta\left(x_n|z_n\right)\, dz_n \\ &= \sum_{n=1}^{N} \mathbb{E}_{q_\phi(z_n|x_n)} \nabla_\phi\, \log q_\phi\left(z_n|x_n\right)\log p_\theta\left(x_n|z_n\right)\end{aligned}$$

So now we can sample from q and approximate this gradient with Monte Carlo. Although this approximation works, it is very loosy, since its variance will be very high. To understand the reason behind it, we should focus on the $\log p_\theta\left(x_n|Z\right)$ term. At the start of the training process, our model initializes to every image very low probabilities, since the model supposes that the training images are really improbable. Combing this low number with the logarithm, the result will be exponentially low. Furthermore, one can observe that the first term, $\nabla_\phi \log q_\phi\left(z_n|x_n\right)$, is very likely that it will change sine for every step. Thus, applying Monte Carlo, and average a few samples will resolve in large positive and negative numbers and the variance will be so high, that we will need lots and lots of samples to approximate it accurately.

Training this type of model has been a long-standing problem in the machine learning community, right until the *reparametrization trick* was proposed in [1].

[†] Appears also on [1] appendix.

[‡] *log-derivative trick*, where $\nabla \log g(\phi) = \frac{\nabla g(\phi)}{g(\phi)}$.

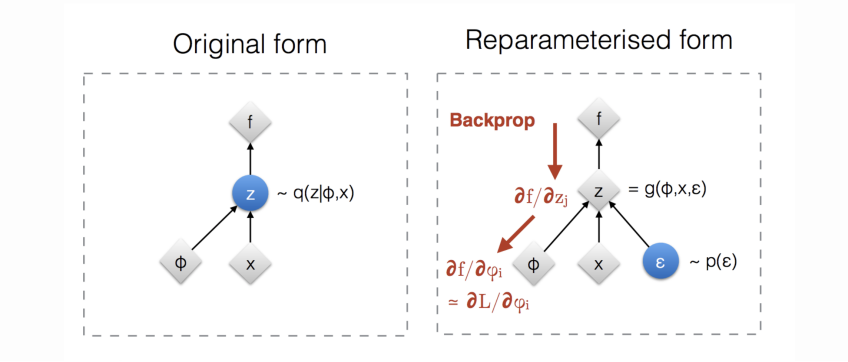

Figure 3: Reparametrization Trick. Blue nodes represent random variable while white deterministic.

c) In [1], Kingma and Welling proposed a simple and brilliant idea on how to resolve it. The reparameterization trick will essentially reduce the variance of the MC estimator for the gradient dramatically. Lets recall that $z_n \sim q_\phi(z_n|x_n) = \mathcal{N}(z_n|\mu_\phi(x_n), \text{diag}(\Sigma_\phi(x_n)))$. Now, instead of sampling z_n from $q_\phi(z_n|x_n)$, we will sample a new *independent* random variable $\varepsilon_n \sim \mathcal{N}(0,1)$, and then sample z_n from the distribution

$$z_i = \varepsilon_i \odot \text{diag}(\Sigma_\phi(x_n)) + \mu_\phi(x_n) = g(\varepsilon_i, x_i, \phi) \quad \text{, where } \varepsilon_n \sim p(\varepsilon_n) = \mathcal{N}(0,1)$$

In this way, the distribution of $q_\phi(z_n|x_n)$ remains the same, it did not change. So now our derivative can be written as

$$\begin{aligned}
\nabla_\phi \sum_{n=1}^{N} \mathbb{E}_{q_\phi(z_n|x_n)}[\log p_\theta(x_n|z_n)] &= \nabla_\phi \sum_{n=1}^{N} \mathbb{E}_{p(\varepsilon_i)}[\log p_\theta(x_n|g(\varepsilon_i, x_i))] \\
&= \sum_{n=1}^{N} \mathbb{E}_{p(\varepsilon_i)} \nabla_\phi \log p_\theta(x_n|g(\varepsilon_i, x_i)) \\
&\approx \sum_{n=1}^{N} \nabla_\phi \log p_\theta(x_n|\widehat{z}_n), \quad \widehat{z}_n \sim q_\phi(z_n|x_n)
\end{aligned}$$

We can now differentiate (backpropagation) w.r.t. the parameters ϕ of the variational distribution using a Monte Carlo approximation with draws from the last distribution.

1.6 Putting things together: Building a VAE

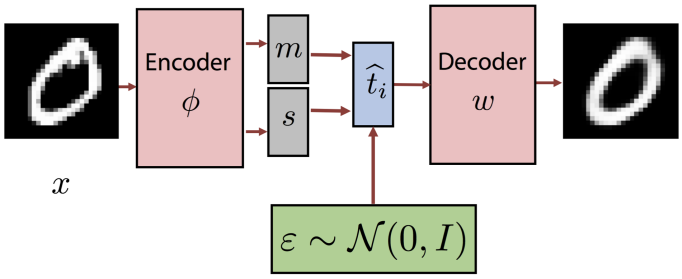

Figure 4: VAE pipeline.

Question 1.12

We implemented a VAE, in order to train it on MNIST dataset and generate digits. In our implementation, as an encoder we used a MLP with two hidden layers of dimension 500 and 250 respectively, with ReLU as non-linearity. The last layer outputs two separate layers, one for the mean value and another for the log-variance, as it will give us better numerical stability. For the decoder, we used the same MLP

structure like the encoder, but reversed. Thus, two hidden layers with ReLU as activation function, and the sigmoid function was used as non-linearity for the output. As a prior distribution, we were using standard normal distribution with mean of 0 and standard deviation of 1. The reparametrization trick was applied to the output of the encoder, in order to sample the latent variable z and perform backpropagation efficiently. Finally, though the regularization error (kl-divergence) and the reconstruction error (binary cross entropy), we calculated the ELBO loss of a pass of the VAE.

Question 1.13

The estimated lower-bounds plot of the VAE training and validation set as training progresses, using a 20-dimensional latent space, can be seen on figure 5. As we can see, the model rapidly moves form 180 to around 100 on epoch 12, and then the learning process is low. This is also reflected on the sample images that appear on the next question.

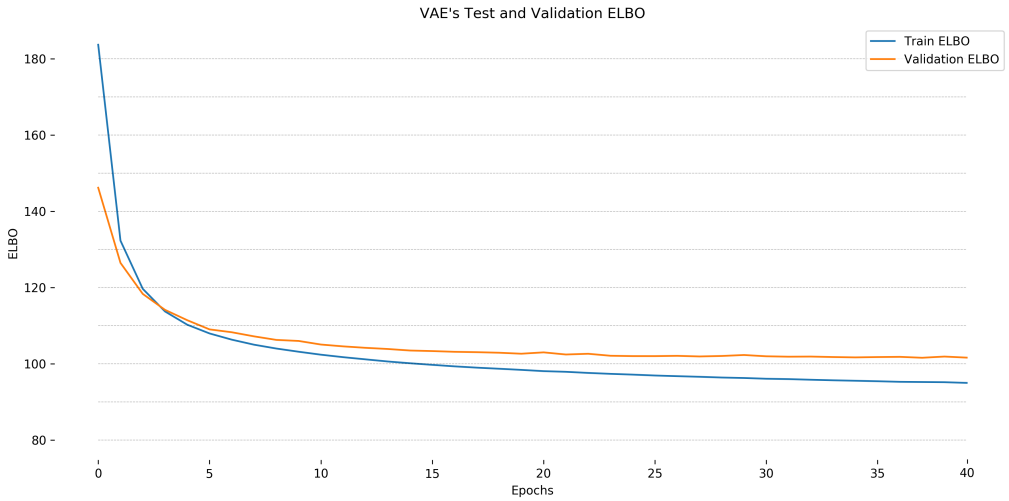

Figure 5: VAE average ELBO loss per epoch.

Question 1.14

We perform experiments both by sampling random noise every time and by initialize a fixed one. The latter will give as a clearer image on the learning proccess of our model.

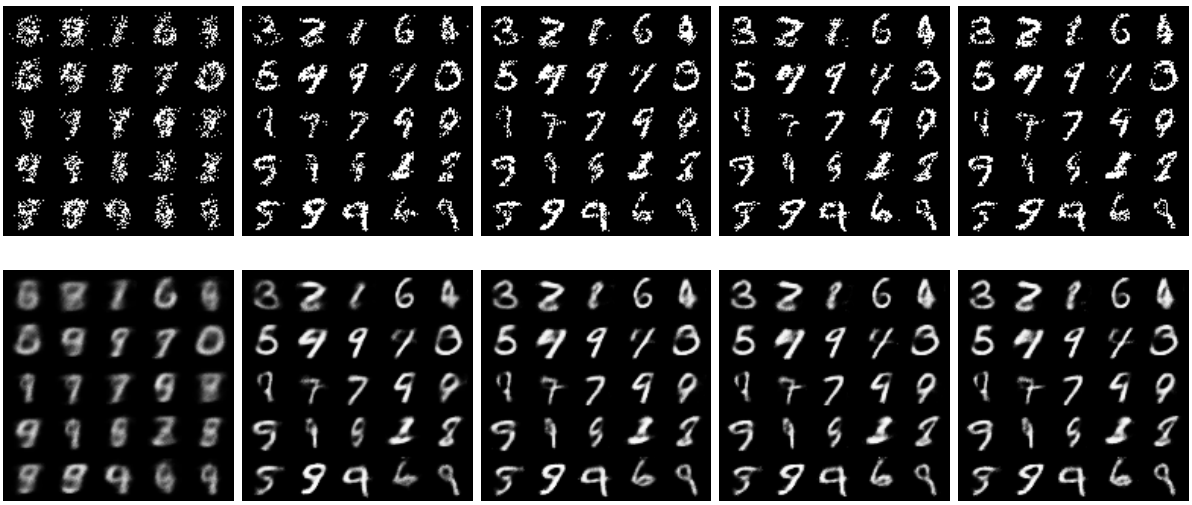

Figure 6: VAE's with 20-dimensional latent space samples from fixed points of the latent space on the 1, 10, 20, 30 and 40 epoch. First row shows the Bernoulli output while the second the mean.

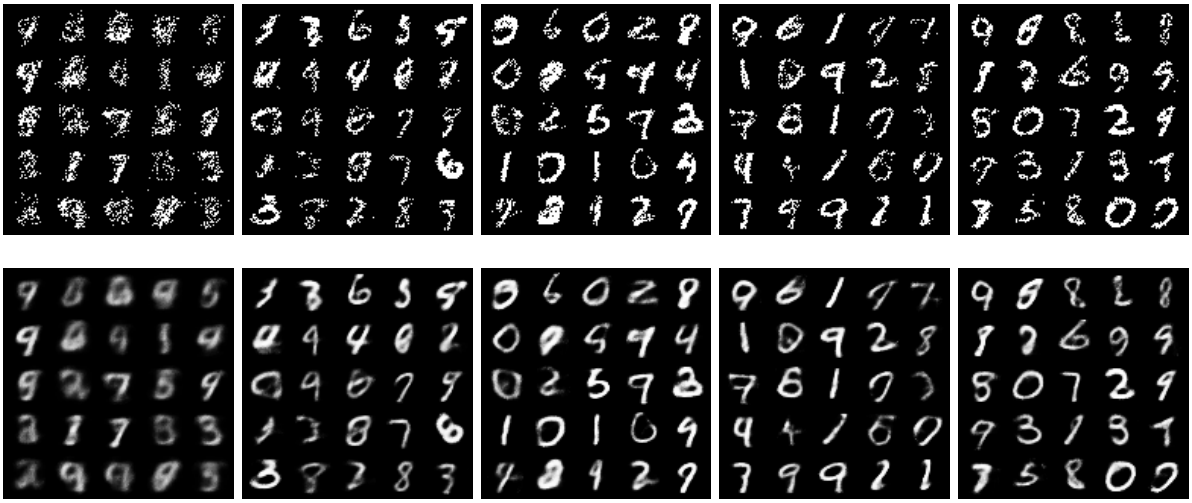

Figure 7: VAE's with 20-dimensional latent space samples from random points of the latent space on the 1, 10, 20, 30 and 40 epoch. First row shows the Bernoulli output while the second the mean.

Question 1.15

VAEs' data manifold is illustrated on figure 8. As we can see, all ten the digits appear and between them there are smooth changes. From these, we can argue that the learned manifold is meaningful.

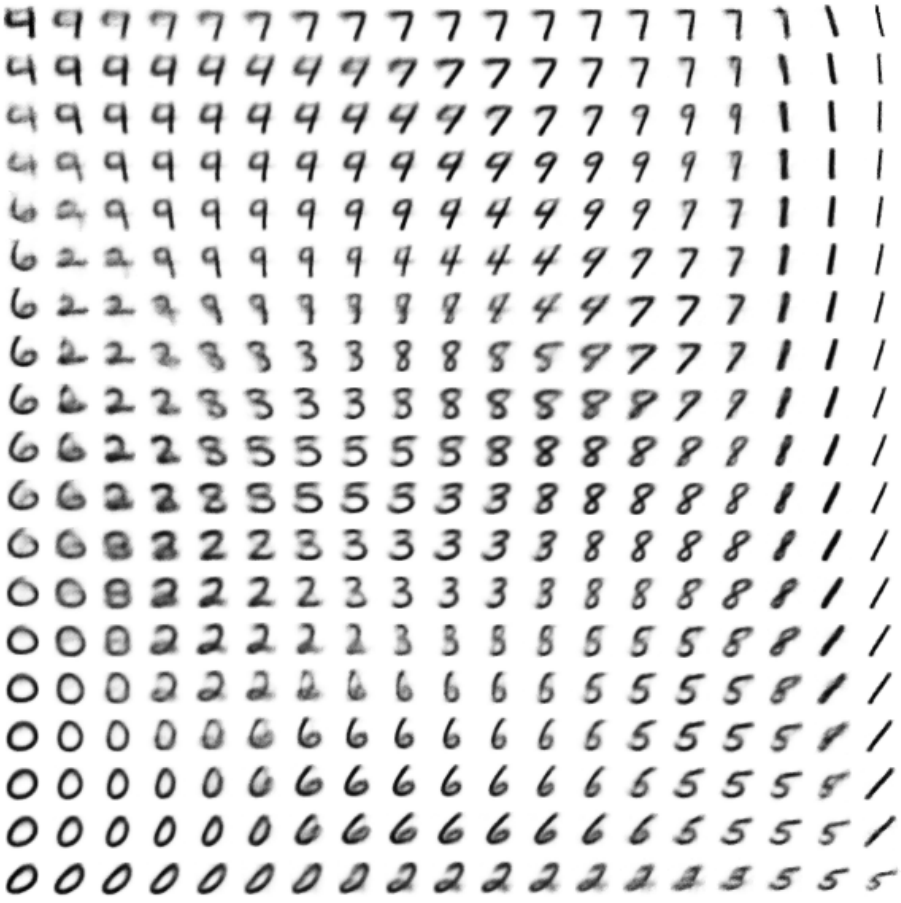

Figure 8: VAE manifold.

2 Generative Adversarial Networks

Question 2.1

A **Generative Adversarial Networks** (GAN) is a generative model that learns to estimate the underlying probability distribution of the data. It consists of two models: A **generator**, that outputs synthetic samples given a noise variable input z, and a **discriminator**, that is optimized to distinguish real data from fake ones. In order the latter model to be trained, it takes as inputs both samples from the dataset (real data), and fake data generated by the generator, outputting a probability of the data being real and not. This pipeline is illustrated on figure 9.

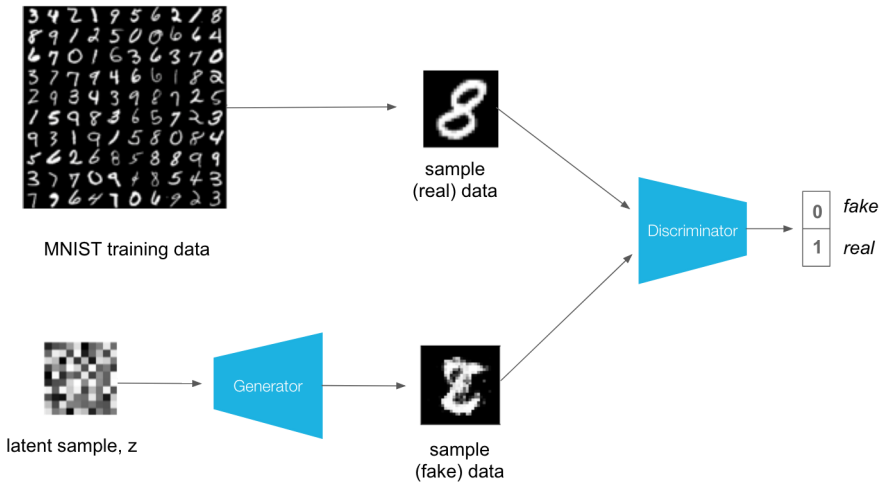

Figure 9: GAN Pipeline

2.1 Training objective: A Minimax Game

Question 2.2

The GAN training objective is defined by the equation

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

Taking it term by term, we have

- $\mathbb{E}_{p_{\text{ data }}(x)}[\log D(X)]$: Expected Log likelihood of discriminator output when the samples from original data distribution are passed as input. As the discriminator represents the probability that the input image is real, this term has be maximized.
- $\mathbb{E}_{p_z(z)}[\log(1 - D(G(Z)))]$: Expected Log likelihood of discriminator output when the samples from generator's output are passed onto discriminator. On one hand, the generator has to maximize the chances of the discriminator getting fooled by the generated images, the former wants to maximize the term $D(G(Z))$. On the other hand, given a fake sample the discriminator is expected to output a probability $D(G(z))$ close to zero by maximizing this term. Thus, this term, according to the generator, has to be minimized and to the discriminator, to be maximized.

When combining both aspects together, the discriminator and the generator are playing a **mini-max game** in which the latter tries to make data as real as the ground truth ones, and the former is trying to be as good as at distinguish the fake from the real ones.

Question 2.3

Let first define with P_z the distribution of noise z, P_g the distribution of generated samples and finally the distribution of real samples as P_r.

Because $E_{x\sim p(x)}[f(x)] = \int_x p(x)f(x)dx$, we can rewrite the loss function as

$$\begin{aligned}\min_G \max_D V(D,G) &= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))] \\ &= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1-D(x)] \\ &= \int_x {}_r(x)\log D(x) + p_g(x)\log(1-D(x))dx\end{aligned}$$

If G is fixed, the optimal Discriminator D^* is (we can safely ignore the integral because x is sampled over all the possible values):

$$\frac{\partial}{\partial D(x)}{}_r(x)\log D(x) + p_g(x)\log(1-D(x)) = D^*(x) = \frac{p_r(x)}{p_r(x)+p_g(x)} \tag{0.6}$$

The global optimal happens when the distribution of generated samples is the same with the distribution of real samples ($p_r = p_g$). This will results to Nash equilibrium, and going back to (0.6), we result in

$$D^*(x) = \frac{1}{2}$$

Thus, when both G and D are at their optimal values, the loss function becomes:

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

Question 2.4

In the beginning of the training, because the discriminator does not have any insides of the data, both samples from the generator and the real ones are equally real to it, thus it picks randomly. This results in providing poor feedback (gradients) to the generator and the learning process is noisy. To resolve this problem, a solution would be to pre-train the discriminator for a small proportion of batches, before introducing the generator into the model.

2.2 Building a GAN

Question 2.5

In our implementation, we first normalize the data to range $[-1, 1]$, in order to have symmetrically distributed data. For the generator, we used a 5-layered MLP with `LeakyReLU` non-linearities with negative slope 0.2. Batch Normalization and dropout of 0.2 were also used, except from the fist layer, that the latent variables are mapped to the first hidden layer. This lead to a tanh activation function in the end. As for the discriminator, a 3-layered MLP with `LeakyReLU` non-linearities with negative slope 0.2 was used, with a sigmoid activation on the output. `Dropout` was used only on the first layer with probability 0.2. We flip and provide additionally noise to the labels, where the labels for the real examples was ranging from 0 to 0.3 and for the fake from 0.7 to 1.2.

On the training procedure, we sampled at every iteration random latent variables and they where used to train both the discriminator and the generator. Although we tried to sample every time two different samples, the former method was faster and more efficient. We first trained the discriminator on the real data, followed by the fake, before we propagate and update its weights. Finally, the generator was trained though the discriminator on the samples that it generated. This completes one step of the learning process.

Question 2.6

Figure 11 illustrates the progress of samples from the GAN. We can clearly derive that as the training progress, the model produces more and more realistic samples, where the sample of the 200th epoch is almost indistinguishable from the original data. The motivation of producing samples from fixed latent points, is that we can watch the progress of the model in more detail.

Note: We also provide a video (`.mov`) and an animated image (`.gif`) along with the report, that shows the training progress of the model.

Figure 10: GAN samples from fixed points of the latent space. From left to right; 1, 12 50 110 and 200 epoch.

Figure 11: GAN samples from random points of latent space. From left to right; 1, 12 50 110 and 200 epoch.

Question 2.7

On figure 12, we provide four different interpolations between two different digits. Because the one digit interpolates smoothly to another, taking even forms of other digits that their form is somehow similar between this digits, we can conclude that the model has actually learned a meaningful continuous space that we can sample from.

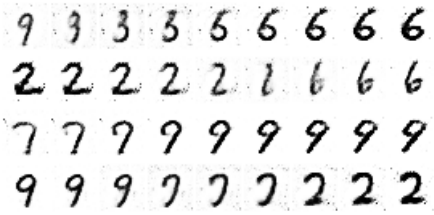

Figure 12: Four examples of GAN interpolations.

3 Generative Normalizing Flows

3.1 Change of variables for Neural Networks

Question 3.1

Analogously to (16), multivariate probability density function of $\mathbf{z}$ can be written as

$$p(\mathbf{x}) = \frac{p(\mathbf{z})}{|J|}$$

where $|J|$ is the absolute value of the Jacobian of the transformation. The Jacobian is the determinant of the matrix of partial derivatives

$$J = \det \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_1}{\partial x_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial z_n}{\partial x_1} & \cdots & \frac{\partial z_n}{\partial x_n} \end{bmatrix} \overset{\S}{=} \det \partial \boldsymbol{z} \frac{1}{\partial \boldsymbol{x}^T} = \det \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}^T}$$

Thus, we can re-write the above as

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

and taking the logarithm on the above equation, we result into

$$\log p(\mathbf{x}) = \log p(\mathbf{z}) + \log \left| \det \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}^T} \right| \tag{0.7}$$

Question 3.2

As we mention, the function f must be *invertible* and *differential*. However, as the dimensions of $\mathbf{z}$ and $\mathbf{x}$ grow, we want the f to have *flexible* invertible transformations (inflexible transformations lead to long sequences of flows for flexible posteriors) and easily computable Jacobian determinants. For the latter, in general, if we have a $D \times D$ matrix, the complexity scale is $O\left(D^3\right)$ and we want to make sure that this actually reduced for invertible transformations.

Question 3.3

Very deep models (many flow steps) need a very big amount of memory and computational power to get the best results. This happens because defining a Jacobian for a neural network is not trivial, neither the determinant of a vary large matrix.

Question 3.4

In [5] they describe very beautifully:

> *"Many real-world datasets, such as CIFAR10 and ImageNet, are recordings of continuous signals quantized into discrete representations. Fitting a continuous density model to discrete data, however, will produce a degenerate solution that places all probability mass on discrete datapoints. A common solution to this problem is to first convert the discrete data distribution into a continuous distribution via a process called* ***dequantization****, and then model the resulting continuous distribution using the continuous density model."*

We understand that modelling discrete datapoints, we will not result into a continuous meaningful latent space, which is our goal for our generative model, in order to have the ability to sample a random point and generate a sample related to the input datapoints. Thus, the idea is to turn this discrete space, lets say {0, 1, 2, ..., 255 } the space of a digital image, to turn it into a continuous one. One simple idea to achieve this, is to divide every value by 256, and result with a [0, 1) continuous space.

[§] The division operation is performed element-wise.

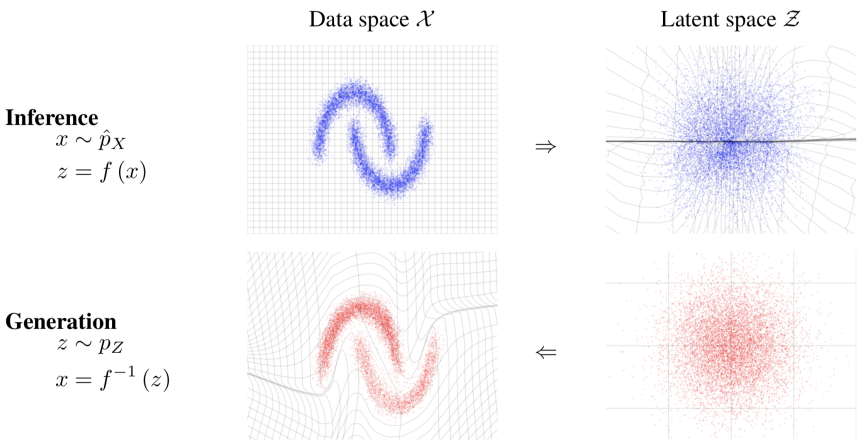

Figure 13: Illustration of the inference and the sampling process of flow-based generative models. Please note the space distortion on the upper right and the lower left plots.

3.2 Building a flow-based model

Question 3.5

One the one hand, when we interesting in inference, we are interesting in the MLE. We make use of the function f, which maps samples $\boldsymbol{x}$ from the data distribution into approximate samples $\boldsymbol{z}$ from the latent distribution. This corresponds to exact inference of the latent state given the data. On the other hand, The inverse function, $f^{-1}(z) = g(z)$, maps samples $\boldsymbol{z}$ from the latent distribution into approximate samples $\boldsymbol{x}$ from the data distribution. This corresponds to exact generation of samples from the model. This process is illustrated on figure 13.

In summary we have that

- **Inference** $P_X(x)$ is the product of $P_Z(f(x))$ and its Jacobian determinant.
- **Sampling** Draw $z \sim P_Z$, and generate a sample $x = f^{-1}(z) = g(z)$.

Question 3.6

In general, the pipeline of training a flow based model works as follows; We sample $\boldsymbol{x} = \boldsymbol{h}_0$ form the true distribution and we feed it as an input to a **bijection**, an invertible and differential function $f : X \to Y$. This will result into a new random variable $f_1(\boldsymbol{x}) = \boldsymbol{h}_1$, where $\boldsymbol{h}_1 \sim p_1(\boldsymbol{h}_1) = p_X(\mathbf{x}) \left|\det\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{h}_1^T}\right)\right|$, and the determinant describes the change of the density between the two pdfs. This process is defined as **flow**. Then, applying f on $\boldsymbol{h}_1$, we get $f_2(\boldsymbol{h}_1) = \boldsymbol{h}_2$, where

$$\boldsymbol{h}_2 \sim p_2(\boldsymbol{h}_2) = p_1(\mathbf{h}_1)\left|\det\left(\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{h}_2^T}\right)\right| = p_X(\mathbf{x})\left|\det\left(\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{h}_1^T}\right)\right|\left|\det\left(\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{h}_2^T}\right)\right|$$

We continue this process for L flows, ending up to the latent space Z. This path which is traversed by the full chain formed by the successive distributions p_i is called a **normalizing flow** and we can illustrate it as follows

$$\mathbf{x} = \mathbf{h}_0 \overset{\mathbf{f}_1}{\longleftrightarrow} \boldsymbol{h}_1 \overset{\mathbf{f}_2}{\longleftrightarrow} \boldsymbol{h}_2 \cdots \overset{\mathbf{f}_L}{\longleftrightarrow} \mathbf{h}_L = \mathbf{z}$$

Looking back to the equation (0.7), the log likelihood probability of X, can be calculated by

$$\begin{aligned}\log p(\mathbf{x}) &= \log p(\boldsymbol{h}_1) + \log \left|\det \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{x}^T}\right| \\ &= \log p(\boldsymbol{h}_2) + \log \left|\det \frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{h}_1^T}\right| + \log \left|\det \frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{x}^T}\right| \\ &= \dots \\ &= \log p(\boldsymbol{h}_L) + \sum_{i=1}^{L} \log \left|\det \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{h}_{i-1}^T}\right|\end{aligned}$$

where $\boldsymbol{h}_L = \boldsymbol{z}$ and more specific

$$\begin{aligned}\boldsymbol{h}_L &= f_L(\boldsymbol{h}_{L-1}) \\ &= f_L(f_{L-1}(\boldsymbol{h}_{L-2})) \\ &= \dots \\ &= f_L \circ f_{L-1} \circ \cdots \circ f_1\left(\mathbf{h}_0\right) \\ &= f_L \circ f_{L-1} \circ \cdots \circ f_1\left(\mathbf{x}_0\right)\end{aligned}$$

Thus, our objective function now is the **negative log-likelihood** ($-\log p(\mathbf{x})$), which we want to minimize. From the adobe, it is clear that for inference we want to keep track of the changes of the density between the flows (save the determinant on each flow), but also the results of the function f of every flow.

As an algorithm prospective, for the training phase we have to follow the next steps

- We first sample a batch from the ground truth data and we if they are recordings of continuous signals quantized into discrete representation, we dequantizate them.
- Next, as we describe above, we are going from float to float while passing the resulted random variable to the next and the accumulated determinant. The result of the last flow plus the accumulated derivatives would be the log-likelihood.
- We can define out loss as the mean of the negative log-likelihood and back-propagate according to this. Due to exploding gradients, it will be useful (and maybe necessary) to clip the gradients up a certain values (for example 100).

Question 3.7

We implemented a flow-based generative model, in particularly the **Real NVP**, which uses an architecture that even f is constrained, in order to make the processes of inference and sampling tractable, it allows the approximation of every flow to be arbitrary complex (e.g. a deep neural network). This achieved by defying f as a **coupling layer**; conditionally linear mapping with diagonal weights. In this way, the hole architecture is going to be bijective, and regardless of the functions t and s, we will have a tractable Jacobian and easily inverse (from sampling). In mathematical terms, a forward pass can be described by the following equations:

$$y = \begin{cases} y_{1:d} & = x_{1:d} \\ y_{d+1:D} & = x_{d+1:D} \odot \exp\left(s\left(x_{1:d}\right)\right) + t\left(x_{1:d}\right) \end{cases} \qquad \frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \operatorname{diag}\left(\exp\left[s\left(x_{1:d}\right)\right]\right) \end{bmatrix}$$

However, one can notice that a coupling layer transforms linearly only one side of the input, leaving the other half unchained. To resolve this, we can compose this coupling layer; put one coupling layer on top on another that chains the part that was left unchained before. Thus, now we have managed to change both sides of the input and still preserve the wanted properties.

We implemented a Real NVP model and test its performance on a palindrome prediction task.

* Implemented on `a3_nf_template.py`.

Question 3.8

Figures 14 and 15 show results of the Real NVP model on MNIST dataset. As we can see, the model quickly drops below 2.0 and the the learning process becomes slow. Looking at the resulting samples from different epochs, we can comment that even that we can understand that the model is learning and we may manage to distinguish some figures, in general the overall result is blurry. However, increasing the number of flows, having to different neural networks for the scale and transpose parameters, may yield better results.

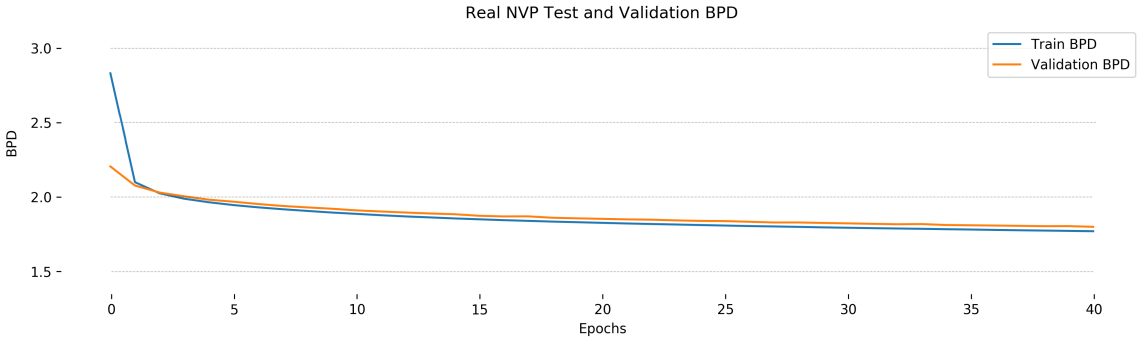

Figure 14: Normalizing Flows loss in BPD over 40 epochs.

Figure 15: Normalizing flows based generative model samples. From left to right; 1, 10 20 30 and 40 epoch.

4 Conclusion

Question 4.1

In this project, we analyze from theoretical point of view and implemented three important generative models. Table 1 sums up the most important consepts of this models.

	VAE	**GANs**	**Read NVP**
Cost	*ELBO (doubly stochastic training) (stable)*	*Approximate adversarial loss (unstable)*	*Log-likelihood (stable)*
Latent space	*Meaningful (Component collapsing)*	*Meaningful (Low-dimensional)*	*Meaningful (High-dimensional)*
Architecture	*Arbitrary architecture and data space (Mean-field inference; until recent work)*	*Arbitrary (Requires continuous data space)*	*Requires partitioning (Requires continuous data space)*

Table 1: Generative models with Deep Neural Networks.

As we saw, VAEs are generative models that produce a meaningful latent space by approximating (lower-bounding) the log-likelihood with the ELBO. Even though they have a nice mathematical framework,

the main drawback is believed to be that the exploit the KL-diverge, which is unsymmetrical. Even though for the experiments we used binary-based MNIST dataset, the results where overall very nice, with the manifold giving us the opportunity to grasp the idea how good they exploit the latent space. GANs, however, are optimizing the JS-divergence, a "symmetrical KL-diverge" distance. Their concept is different from the other two, as they use a minimax game to train two models and generate results like the input dataset. The sample results are the best yet, as one could say that they are almost indistinguishable form the real ones and, as we saw with the interpolation, they manage to learn a meaningful latent space. However, since their cost function is unstable, they are hard to be trained and to understand the learning process. Finally, flow-based generative models are the only one's that model the log-likelihood directly. This make them very powerful, since now the cost function is the actual log-likelihood of the data and the latent space can be high-dimensional. They coming with a strong mathematical framework, but they tend to be slower, since they require a lot of memory for the learning process. Our results do not fully exploit the power of these models, since resent research, shows that they have the ability to produce very sharp images, very close to those produced by GANs.

Bibliography

[1] Auto-Encoding Variational Bayes, Diederik P. Kingma & Max Welling (link).

[2] Tutorial on Variational Autoencoders, Carl Doersch (link).

[3] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In Advances in neural information processing systems, pages 2672–2680, 2014.

[4] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using Real NVP. International Conference on Learning Representations, ICLR, 2017.

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

[6] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows. In Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15, pages 1530–1538. JMLR.org, 2015.

Appendix

Kullback-Leibler divergence

The Kullback-Leibler (KL) divergence is

$$\int_{-\infty}^{\infty} f(x) \log \frac{f(x)}{g(x)} \mathrm{d}x$$

From the hypothesis we have

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad \& \quad g(x) = \frac{1}{\sqrt{2\pi\tau^2}} e^{-\frac{(x-\nu)^2}{2\tau^2}}$$

We begin with expanding the log expression inside the KL

$$\log \frac{f(x)}{g(x)} = \log \frac{\frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\frac{1}{\sqrt{2\pi\tau^2}} e^{\frac{(x-\nu)^2}{2\tau^2}}} = \log \frac{\tau}{\sigma} \frac{e^{\frac{(x-\nu)^2}{2\tau^2}}}{e^{\frac{(x-\mu)^2}{2\sigma^2}}} = \log \frac{\tau}{\sigma} e^{\frac{(x-\nu)^2}{2\tau^2} - \frac{(x-\mu)^2}{2\sigma^2}}$$

$$= \log \frac{\tau}{\sigma} + \log e^{\frac{(x-\nu)^2}{2\tau^2}} - \frac{(x-\mu)^2}{2\sigma^2} = \log \frac{\tau}{\sigma} + \frac{(x-\nu)^2}{2\tau^2} - \frac{(x-\mu)^2}{2\sigma^2}$$

If we put this outcome into the KL we will have

$$\begin{aligned}
\int_{-\infty}^{\infty} f(x)\log\frac{f(x)}{g(x)}\mathrm{d}x &= \int_{-\infty}^{\infty} f(x)\left(\log\frac{\tau}{\sigma}+\frac{(x-\nu)^2}{2\tau^2}-\frac{(x-\mu)^2}{2\sigma^2}\right)\mathrm{d}x \\
&= \int_{-\infty}^{\infty} f(x)\log\frac{\tau}{\sigma}+f(x)\frac{(x-\nu)^2}{2\tau^2}-f(x)\frac{(x-\mu)^2}{2\sigma^2}\mathrm{d}x \\
&= \int_{-\infty}^{\infty} f(x)\log\frac{\tau}{\sigma}\mathrm{d}x+\int_{-\infty}^{\infty} f(x)\frac{(x-\nu)^2}{2\tau^2}\mathrm{d}x-\int_{-\infty}^{\infty} f(x)\frac{(x-\mu)^2}{2\sigma^2}\mathrm{d}x \\
&= \log\frac{\tau}{\sigma}\int_{-\infty}^{\infty} f(x)\mathrm{d}x+\frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)(x-\nu)^2\mathrm{d}x-\frac{1}{2\sigma^2}\int_{-\infty}^{\infty} f(x)(x-\mu)^2\mathrm{d}x \\
&= \log\frac{\tau}{\sigma}*1+\frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)(x-\nu)^2\mathrm{d}x-\frac{1}{2\sigma^2}\left(\sigma^2\right) \\
&= \log\frac{\tau}{\sigma}+\frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)(x-\nu)^2\mathrm{d}x-\frac{1}{2}
\end{aligned}$$

We will expand the middle part of the last expression

$$\begin{aligned}
\frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)(x-\nu)^2\mathrm{d}x &= \frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)((x-\mu)+(\mu-\nu))^2\mathrm{d}x \\
&= \frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)(x-\mu)^2\mathrm{d}x+\frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)(\mu-\nu)^2\mathrm{d}x \\
&+ \frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)2(x-\mu)(\mu-\nu)\mathrm{d}x \\
&= \frac{1}{2\tau^2}(\sigma)^2+(\mu-\nu)^2\frac{1}{2\tau^2}\int_{-\infty}^{\infty} f(x)\mathrm{d}x+\frac{(\mu-\nu)}{\tau^2}\int_{-\infty}^{\infty} f(x)(x-\mu)\mathrm{d}x
\end{aligned}$$

Taking the last part of the previous expression:

$$\int_{-\infty}^{\infty} f(x)(x-\mu)\mathrm{d}x = \int_{-\infty}^{\infty} f(x)x\mathrm{d}x-\int_{-\infty}^{\infty} f(x)\mu\mathrm{d}x = \mu-1*\mu = 0$$

Thus

$$\frac{\sigma^2}{2\tau^2}+\frac{(\mu-\nu)^2}{2\tau^2}*1+0 = \frac{\sigma^2}{2\tau^2}+\frac{(\mu-\nu)^2}{2\tau^2}$$

And therefore it is

$$\log\frac{\tau}{\sigma}+\frac{\sigma^2}{2\tau^2}+\frac{(\mu-\nu)^2}{2\tau^2}-\frac{1}{2}$$

or

$$\frac{1}{2}\left(2\log\frac{\tau}{\sigma}+\frac{\sigma^2+(\mu-\nu)^2}{\tau^2}-1\right) = \frac{1}{2}\left(\log\frac{\tau^2}{\sigma^2}+\frac{\sigma^2+(\mu-\nu)^2}{\tau^2}-1\right)$$