Practical 3. Deep Generative Models

University of Amsterdam – Deep Learning Course

December 2, 2020

The deadline for this assignment is December 20th at 23:59.

Modelling distributions in high dimensional spaces is difficult. Simple distributions such as multivariate Gaussians or mixture models are not powerful enough to model complicated high-dimensional distributions. The question is: How can we design complicated distributions over high-dimensional data, such as images or audio? In concise notation, how can we model a distribution $p(\boldsymbol{x}) = p(x_1, x_2, \ldots, x_M)$, where M is the number of dimensions of the input data $\boldsymbol{x}$? The solution: Deep Generative Models.

Deep generative models come in many flavors, but all share a common goal: to model the probability distribution of the data. Examples of well-known generative models are Variational Autoencoders (VAEs), Generative Adversarial Networks (GANs) and Normalizing Flows (NFs or flows). VAEs can evaluate a lower bound on the likelihood and GANs can typically only draw samples. On the other hand, Normalizing Flows can evaluate the exact likelihood of the model distribution.

In this assignment, we will focus on the above mentioned models: VAEs [3], GANs [2] and Normalizing Flows [4]. You will implement them in PyTorch as part of this assignment. Note that, although this assignment does contain some explanation on the model, we do not aim to give a complete introduction. The best source for understanding the models are the lectures, these papers [2, 3, 4], and the hundreds of blog-posts that have been written on them ever since.

Throughout this assignment you will see a new type of boxes between questions, namely **Food for thought** boxes. Those contain questions that are helpful for understanding the material, but are not essential and **not required to submit in the report** (no points are assigned to those question). Still, try to think of a solution for those boxes to gain a deeper understanding of the models.

This assignment contains 100 points: 45 on VAEs, 30 on GANs, 20 on Flow-Based methods and 5 for Conclusion. We provide a time estimate for each so that you plan of how much work you can expect from each section.

Note: for this assignment you are *not* allowed to use the `torch.distributions` package. You are, however, allowed to use standard, stochastic PyTorch functions like `torch.randn` and `torch.binomial`, and all other PyTorch functionalities (especially from `torch.nn`). Moreover, try to stay as close as your can to the template files provided as part of the assignment.

1 Variational Auto Encoders **(Total: 45 points)**

Time estimate: 12 hours (7 hours theory, 5 hours implementation+experiments)

VAEs leverage the flexibility of neural networks (NN) to learn and specify a latent variable model. We will first briefly discuss Latent Variable Models and then dive into VAEs. Mathematically, they are connected to a distribution $p(\boldsymbol{x})$ over $\boldsymbol{x}$ in the following way: $p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})d\boldsymbol{z}$. This integral is typically too expensive to evaluate, which VAEs have resolved in a way that you learn in this assignment.

1.1 Latent Variable Models

A latent variable model is a statistical model that contains both observed and unobserved (i.e. latent) variables. Assume a dataset $\mathcal{D} = \{\boldsymbol{x}_n\}_{n=1}^N$, where $\boldsymbol{x}_n \in \{0,1\}^M$. For example, $\boldsymbol{x}_n$ can be the pixel values of a binary image. A simple latent variable model for this data is shown in Figure 1, which we can also summarize with the following generative story:

$$\boldsymbol{z_n} \sim \mathcal{N}(0, \boldsymbol{I}_D) \tag{1}$$
$$\boldsymbol{x_n} \sim p_X(f_\theta(\boldsymbol{z}_n)) \tag{2}$$

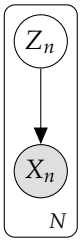

Figure 1. Graphical model of VAE. N denotes the dataset size.

where f_θ is some function – parameterized by θ – that maps $\boldsymbol{z}_n$ to the parameters of a distribution over $\boldsymbol{x}_n$. For example, if p_X would be a Gaussian distribution we will use $f_\theta : \mathbb{R}^D \to (\mathbb{R}^M, \mathbb{R}_+^M)$ for a mean and covariance matrix, or if p_X is a product of Bernoulli distributions, we have $f_\theta : \mathbb{R}^D \to [0,1]^M$. Here, D denotes the dimensionality of the latent space. Note that our dataset $\mathcal{D}$ does not contain $\boldsymbol{z}_n$, hence $\boldsymbol{z}_n$ is a latent (or unobserved) variable in our statistical model. In the case of a VAE, a (deep) NN is used for $f_\theta(\cdot)$.

Food for thought

How does the VAE relate to a standard autoencoder (see e.g. Tutorial 9)?

1. Are they different in terms of their main purpose? How so?
2. A VAE is generative. Can the same be said of a standard autoencoder? Why or why not?
3. Can a VAE be used in place of a standard autoencoder for its purpose you mentioned above?

1.2 Decoder: The Generative Part of the VAE

In the previous section, we described a general graphical model which also applies to VAEs. In this section, we will define a more specific generative model that we will use throughout this assignment. This will later be refered to as the *decoding* part (or decoder) of a VAE. For this assignment we will assume the pixels of our images $\boldsymbol{x}_n$ in $\mathcal{D}$ are Bernoulli(p) distributed.

$$p(\boldsymbol{z}_n) = \mathcal{N}(0, \boldsymbol{I}_D) \tag{3}$$
$$p(\boldsymbol{x}_n|\boldsymbol{z}_n) = \prod_{m=1}^{M} \text{Bern}(\boldsymbol{x}_n^{(m)}|f_\theta(\boldsymbol{z}_n)_m) \tag{4}$$

where $\boldsymbol{x}_n^{(m)}$ is the m-th pixel of the n-th image in $\mathcal{D}$, and $f_\theta : \mathbb{R}^D \to [0,1]^M$ is a neural network parameterized by θ that outputs the means of the Bernoulli distributions for each pixel in $\boldsymbol{x}_n$.

Question 1.1 (3 points)

Describe the steps needed to *sample* from such a model. (Hint: ancestral sampling)

Food for thought

This model makes a very simplistic assumption about $p(Z)$, i.e. it assumes that our latent variables follow a standard-normal distribution. Note, that there is no trainable parameter in $p(Z)$. Describe why, due to the nature of Equation 4, this is not such a restrictive assumption in practice. (Hint: See Figure 1 and the accompanying explanation in Carl Doersch's tutorial)

Now that we have defined the model, we can write out an expression for the log probability of the data $\mathcal{D}$ under this model:

$$\begin{aligned}\log p(\mathcal{D}) &= \sum_{n=1}^{N} \log p(\boldsymbol{x}_n) \\ &= \sum_{n=1}^{N} \log \int p(\boldsymbol{x}_n|\boldsymbol{z}_n)p(\boldsymbol{z}_n)d\boldsymbol{z}_n \\ &= \sum_{n=1}^{N} \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right]\end{aligned} \tag{5}$$

Evaluating $\log p(\boldsymbol{x}_n) = \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right]$ involves a very expensive integral. However, Equation 5 hints at a method for approximating it, namely Monte-Carlo Integration. The log-likelihood can be approximated by drawing samples $\boldsymbol{z}_n^{(l)}$ from $p(\boldsymbol{z}_n)$:

$$\log p(\boldsymbol{x}_n) = \log \mathbb{E}_{p(\boldsymbol{z}_n)}\left[p(\boldsymbol{x}_n|\boldsymbol{z}_n)\right] \tag{6}$$

$$\approx \log \frac{1}{L}\sum_{l=1}^{L} p(\boldsymbol{x}_n|\boldsymbol{z}_n^{(l)}), \quad \boldsymbol{z}_n^{(l)} \sim p(\boldsymbol{z}_n) \tag{7}$$

If we increase the number of samples L to infinity, the approximation would be equals to the actual expectation. Hence, the estimator is unbiased and can be used to approximate $\log p(\boldsymbol{x}_n)$ with a sufficient large number of samples.

Question 1.2 (3 points)

Although Monte-Carlo Integration with samples from $p(\boldsymbol{z}_n)$ can be used to approximate $\log p(\boldsymbol{x}_n)$, it is not used for training VAE type of models, because it is inefficient. In a few sentences, describe why it is inefficient and how this efficiency scales with the dimensionality of $\boldsymbol{z}$. (Hint: you may use Figure 2 in you explanation.)

1.3 KL Divergence

Before continuing our discussion about VAEs, we will need to learn about another concept that will help us later: the Kullback-Leibler divergence (KL divergence). It measures how different one probability distribution is from another:

$$D_{\mathrm{KL}}(q||p) = -\mathbb{E}_{q(x)}\left[\log \frac{p(X)}{q(X)}\right] = -\int q(x)\left[\log \frac{p(x)}{q(x)}\right]dx, \tag{8}$$

where q and p are probability distributions in the space of some random variable X.

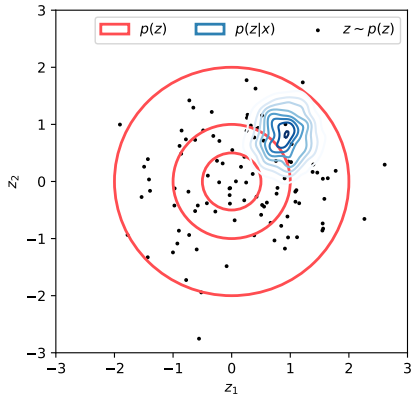

Figure 2. Plot of 2-dimensional latent space and contours of prior and posterior distributions.

Question 1.3 (2 points)

Assume that q and p in Equation 8, are univariate gaussians: $q = \mathcal{N}(\mu_q, \sigma_q^2)$ and $p = \mathcal{N}(\mu_p, \sigma_p^2)$. Give two examples of $(\mu_q, \mu_p, \sigma_q^2, \sigma_p^2)$: one of which results in a very small, and one of which has a very large, KL-divergence: $D_{\mathrm{KL}}(q||p)$.

In VAEs, we usually set the prior to be a normal distribution with a zero mean and unit variance: $p = \mathcal{N}(0, 1)$. For this case, we can actually find a closed-form solution of the KL divergence:

$$KL(q, p) = -\int q(x) \log p(x) dx + \int q(x) \log p(x) dx \tag{9}$$

$$= \frac{1}{2} \log(2\pi\sigma_p^2) + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} - \frac{1}{2}(1 + \log 2\pi\sigma_q^2) \tag{10}$$

$$= \log \frac{\sigma_p}{\sigma_q} + \frac{\sigma_q^2 + (\mu_q - \mu_p)^2}{2\sigma_p^2} - \frac{1}{2} \tag{11}$$

$$= \frac{\sigma_q^2 + \mu_q^2 - 1 - \log \sigma_q^2}{2} \tag{12}$$

For simplicity, we skipped a few steps in the derivation. You can find the details here if you are interested (it is not essential for understanding the VAE). We will need this result for our implementation of the VAE later.

1.4 The Encoder: $q_\phi(z_n|x_n)$ - Efficiently evaluating the integral

In the previous section 1.2, we have developed the intuition that we only want to sample z_n for which $p(z_n|x_n)$ is not close to zero – in order to compute the Monte-Carlo approximation. Unfortunately, the true posterior $p(z_n|x_n)$ is as difficult to compute as $p(x_n)$ itself. To solve this problem, instead of modeling the true posterior $p(z_n|x_n)$, we can learn an *approximate posterior distribution*, which we refer to as the *variational distribution*. This variational distribution $q(z_n|x_n)$ is used to approximate the (very expensive) posterior $p(z_n|x_n)$ and to more efficiently integrate $\int p(x_n|z_n)p(z_n)dz_n$.

Now we have all the tools to derive an efficient bound on the log-likelihood $\log p(\mathcal{D})$. We start from Equation 5 where the log-likelihood objective is written, but for simplicity in

notation we write the log-likelihood $\log p(\boldsymbol{x}_n)$ only for a single datapoint.

$$
\begin{aligned}
\log p(\boldsymbol{x}_n) &= \log \mathbb{E}_{p(\boldsymbol{z}_n)} \left[p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] \\
&= \log \mathbb{E}_{p(\boldsymbol{z}_n)} \left[\frac{q(\boldsymbol{z}_n|\boldsymbol{x}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] \quad \text{(multiply by } q(\boldsymbol{z}_n|\boldsymbol{x}_n)/q(\boldsymbol{z}_n|\boldsymbol{x}_n)\text{)} \\
&= \log \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \left[\frac{p(\boldsymbol{z}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] \quad \text{(switch expectation distribution)} \\
&\geq \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \log \left[\frac{p(\boldsymbol{z}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] \quad \text{(Jensen's inequality)} \\
&= \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \left[\log p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] + \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \log \left[\frac{p(\boldsymbol{z}_n)}{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \right] \quad \text{(re-arranging)} \\
&= \underbrace{\mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \left[\log p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] - KL\left(q(Z|\boldsymbol{x}_n)||p(Z)\right)}_{\text{Expected Lower Bound (ELBO)}} \quad \text{(writing 2nd term as KL)}
\end{aligned}
\tag{13}
$$

This is awesome! We have derived a bound on $\log p(\boldsymbol{x}_n)$, exactly the thing we want to optimize, where all terms on the right hand side are computable. Let's put together what we have derived again in a single line:

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

The right side of the equation is referred to as the *evidence lowerbound* (ELBO) on the log-probability of the data.

This leaves us with the question: How close is the ELBO to $\log p(\boldsymbol{x}_n)$? With an alternate derivation[1] we can find the answer. It turns out the gap between $\log p(\boldsymbol{x}_n)$ and the ELBO is exactly $KL(q(Z|\boldsymbol{x}_n)||p(Z|\boldsymbol{x}_n))$ such that:

$$\log p(\boldsymbol{x}_n) - KL(q(Z|\boldsymbol{x}_n)||p(Z|\boldsymbol{x}_n)) = \mathbb{E}_{q(\boldsymbol{z}_n|\boldsymbol{x}_n)} \left[\log p(\boldsymbol{x}_n|\boldsymbol{z}_n) \right] - KL\left(q(Z|\boldsymbol{x}_n)||p(Z)\right) \tag{14}$$

Now, let's optimize the ELBO. For this, we define our loss as the mean negative lower bound over samples:

$$\mathcal{L}(\theta, \phi) = -\frac{1}{N} \sum_{n=1}^{N} \mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|Z)] - D_{\mathrm{KL}}(q_\phi(Z|\boldsymbol{x}_n)||p_\theta(Z)) \tag{15}$$

Note, that we make an explicit distinction between the generative parameters θ and the variational parameters ϕ.

Question 1.4 (4 points)

Explain how you can see from Equation 14 that the right hand side has to be a *lower bound* on the log-probability $\log p(\boldsymbol{x}_n)$? Why must we optimize the lower-bound, instead of optimizing the log-probability $\log p(\boldsymbol{x}_n)$ directly?

Question 1.5 (3 points)

Now, looking at the two terms on left-hand side of 14: Two things can happen when the lower bound is pushed up. Can you describe what these two things are?

[1] This derivation is not done here, but can be found in for instance Bishop sec 9.4.

1.5 Specifying the Encoder $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$

In VAE, we have some freedom to choose the distribution $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$. In essence, we want to choose something that can closely approximate $p(\boldsymbol{z}_n|\boldsymbol{x}_n)$, but we are also free to a select distribution that makes our life easier. We will do exactly that in this case and choose $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ to be a factored multivariate normal distribution, i.e.,

$$q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n) = \mathcal{N}(\boldsymbol{z}_n|\mu_\phi(\boldsymbol{x}_n), \text{diag}(\Sigma_\phi(\boldsymbol{x}_n))), \tag{16}$$

where $\mu_\phi : \mathbb{R}^M \to \mathbb{R}^D$ maps an input image to the mean of the multivariate normal over $\boldsymbol{z}_n$ and $\Sigma_\phi : \mathbb{R}^M \to \mathbb{R}^D_+$ maps the input image to the diagonal of the covariance matrix of that same distribution. Moreover, $\text{diag}(\boldsymbol{v})$ maps a K-dimensional (for any K) input vector $\boldsymbol{v}$ to a $K \times K$ matrix such that for $i, j \in \{1, \ldots, K\}$

$$\text{diag}(\boldsymbol{v})_{ij} = \begin{cases} v_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}. \tag{17}$$

Question 1.6 (3 points)

The loss in Equation 15:

$$\mathcal{L}(\theta, \phi) = -\frac{1}{N}\sum_{n=1}^{N} \mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|Z)] - D_{\text{KL}}(q_\phi(Z|\boldsymbol{x}_n)||p_\theta(Z))$$

can be rewritten in terms of per-sample losses:

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

where

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

can be seen as a reconstruction loss term and an regularization term, respectively. Explain why the names *reconstruction* and *regularization* are appropriate for these two losses.

(Hint: Suppose we use just one sample to approximate the expectation $\mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[p_\theta(\boldsymbol{x}_n|Z)]$ – as is common practice in VAEs.)

Question 1.7 (4 points)

Now we have defined an objective (Equation 15) in terms of an abstract model and variational approximation, we can put everything together using our model definition (Equation 1 and 2) and definition of $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ (Equation 16), and we can write down a single objective which we can minimize.

Write down expressions (including steps) for $\mathcal{L}_n^{\text{recon}}$ and $\mathcal{L}_n^{\text{reg}}$ such that we can minimize $\mathcal{L} = \sum_{n=1}^{N} \mathcal{L}_n^{\text{recon}} + \mathcal{L}_n^{\text{reg}}$ as our final objective. Make any approximation explicit.

(Hint: look at Equation 9 for $\mathcal{L}_n^{\text{reg}}$.)

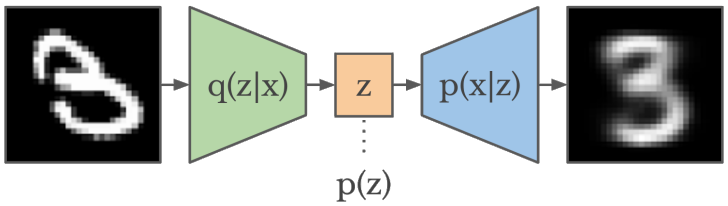

Figure 3. A VAE architecture on MNIST. The encoder distribution $q(z|x)$ maps the input image into latent space. This latent space should follow a unit Gaussian prior $p(z)$. A sample from $q(z|x)$ is used as input to the decoder $p(x|z)$ to reconstruct the image. Figure taken from this blog.

1.6 The Reparametrization Trick

Although we have written down (the terms of) an objective in question 1.7, we still cannot simply minimize this by taking gradients with regard to θ and ϕ. This is due to the fact that we sample from $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ to approximate the $\mathbb{E}_{q_\phi(z|\boldsymbol{x}_n)}[\log p_\theta(\boldsymbol{x}_n|Z)]$ term. Yet, we need to pass the derivative through these samples if we want to compute the gradient of the encoder parameters, i.e., $\nabla_\phi \mathcal{L}(\theta, \phi)$. Our posterior approximation $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ is parameterized by ϕ. If we want to train $q_\phi(\boldsymbol{z}_n|\boldsymbol{x}_n)$ to maximize the lower bound, and therefore approximate the posterior, we need to have the gradient of ϕ with respect to the lower-bound.

Question 1.8 (3 points)

Passing the derivative through samples can be done using the *reparameterization trick*. In a few sentences, explain why the act of sampling usually prevents us from computing $\nabla_\phi \mathcal{L}$, and how the reparameterization trick solves this problem.
(Hint: you can take a look at Figure 4 from the tutorial by Carl Doersch)

1.7 Putting things together: Building a VAE

Given everything we have discussed so far, we now have an objective (the *evidence lower bound* or ELBO) and a way to backpropagate to both θ and ϕ (i.e., the reparametrization trick). Thus, we can now implement a VAE in PyTorch to train on MNIST images. We will model the encoder $q(z|x)$ and decoder $p(x|z)$ by a deep neural network each, and train them to maximize the data likelihood. See Figure 3 for an overview of the components we need to consider in a VAE.

In the code directory `part1`, you can find the templates to use for implementing the VAE. We provide two versions for the training loop: a template in PyTorch Lightning (`train_pl.py`), and a template in plain PyTorch (`train_torch.py`). You can choose which you prefer to implement. **You only need to implement one of the two training loop templates**. If you followed the tutorial notebooks, you might want to give PyTorch Lightning a try as it is less work, more structured and has an automatic saving and logging mechanism. You do not need to be familiar with PyTorch Lightning to the lowest level, but a high-level understanding as from the introduction in Tutorial 5 is sufficient for implementing the template.
You also need to implement additional functions in `utils.py`, and the encoder and decoder in the files `mlp_encoder_decoder.py` and `cnn_encoder_decoder.py`. For the first part, you only need to implement the MLP networks. We specified a recommended architecture to start with, but you are allowed to experiment with your own ideas for the models. For the sake of the assignment, it is sufficient to use the recommended architecture to achieve full points. Use the provided unit tests to ensure the correctness of your implementation. Details on the files can be found in the README of part 1.

As a loss objective and test metric, we will use the bits per dimension score (bpd). Bpd is motivated from an information theory perspective and describes how many bits we would need to encode a particular example in our modeled distribution. You can see it as how many bits we would need to store this image on our computer or send it over a network, if we have given our model. The less bits we need, the more likely the example is in our distribution. Hence, we can use bpd as loss metric to minimize. When we test for the bits per dimension on our test dataset, we can judge whether our model generalizes to new samples of the dataset and didn't in fact memorize the training dataset. In order to calculate the bits per dimension score, we can rely on the negative log-likelihood we got from the ELBO, and change the log base (as bits are binary while NLL is usually exponential):

$$\text{bpd} = \text{nll} \cdot \log_2(e) \cdot \left(\prod_i d_i \right)^{-1}$$

where $d_1, ..., d_K$ are the dimensions of the input excluding any batch dimension. For images, this would be the height, width and channel number. We average over those dimensions in order to have a metric that is comparable across different image resolutions. The nll represents the log-likelihood loss $\mathcal{L}$ from Equation 15 for a single data point. You should implement this function in `utils.py`.

Question 1.9 (12 points)

Build a Variational Autoencoder in the provided templates, and train it on the binarized MNIST dataset. Both the encoder and decoder should be implemented as an MLP. Following standard practice – and for simplicity – you may assume that the number of samples used to approximate the expectation in $\mathcal{L}_n^{\text{recon}}$ is 1. Use a latent space size of `z_dim=20`. Read the provided README to become familiar with the code template.

In your report, provide a short description (no more than 10 lines) of the used architectures for the encoder and decoder, any hyperparameters and your training steps. Additionally, plot the estimated bit per dimension score of the lower bound on the training and validation set as training progresses, and the final test score. You are allowed to take screenshots of a TensorBoard plot if the axes values are clear.

Note: using the default hyperparameters is sufficient to obtain full points. As a reference, your training loss should start slightly above 1 at the very first iteration, and go below 0.16.

Question 1.10 (4 points)

Plot 64 samples (8×8 grid) from your model at three points throughout training (before training, after training 10 epochs, and after training 80 epochs). You should observe an improvement in the quality of samples. Describe shortly the quality and/or issues of the generated images.

Question 1.11 (4 points)

Train a VAE with a 2-dimensional latent space (`z_dim=2` in the code). Use this VAE to plot the data *manifold* as is done in Figure 4b of [3]. This is achieved by taking a two dimensional grid of points in Z-space, and plotting $f_\theta(Z) = \mu|Z$. Use the percent point function (ppf, or the inverse CDF) to cover the part of Z-space that has significant density. Implement it in the function `visualize_manifold` in `utils.py`, and use a grid size of 20. You should be able to spot every digit in the manifold. Are you recognizing any patterns of the positions of the digits?

Question 1.12 (Bonus - 3 points)

In the previous questions, we have used MLPs for modeling the encoder and decoder. Next, implement a convolutional encoder and decoder network in the file `cnn_encoder_decoder.py`. For the architecture, you can use the same as used in Tutorial 9 about Autoencoders. Note that you have to adjust the output shape of the decoder to $1 \times 28 \times 28$ for MNIST. You can do this by adjusting the output padding of the first transposed convolution in the decoder.
Plot the train and validation bits per dimension, and report the test score. Also include samples of the model as in Question 1.10. What differences can you see compared to the MLP networks?

2 Generative Adversarial Networks **(30 points)**

Time estimate: 7 hours (1 hour theory, 6 hours implementation+experiments)

Generative Adversarial Networks (GANs) are another type of deep generative models. Similar to VAEs, GANs can generate images that mimic images from the dataset by sampling an encoding from a noise distribution. In contrast to VAEs, in vanilla GANs there is no inference mechanism to determine an encoding or latent vector that corresponds to a given data point (or image). Figure 4 shows a schematic overview of a GAN. A GAN consists of two separate networks (i.e., there is no parameter sharing or the like) called the generator and the discriminator. Training a GAN leverages an adversarial training scheme. In short, that means that instead of defining a loss function by hand (e.g., cross-entropy or mean squared error), we train a network that acts as a loss function. In the case of a GAN this network is trained to discriminate between real images and fake (or generated) images, hence the name *discriminator*. The discriminator (together with the training data) then serves as a loss function for our *generator* network that will learn to generate images to are similar to those in the training set. Both the generator and discriminator are trained jointly. In this assignment, we will focus on obtaining a generator network that can generate images that are similar to those in the training set.

2.1 Training objective: A Minimax Game

In order to train a GAN we have to decide on a noise distribution $p(\boldsymbol{z})$, in this case we will use a standard normal distribution. Given this noise distribution, the GAN training procedure is a minimax game between the generator and discriminator. This is best seen by inspecting the loss (or optimization objective):

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{p_{\text{data}}(\boldsymbol{x})}[\log D(X)] + \mathbb{E}_{p_z(\boldsymbol{z})}[\log(1 - D(G(Z)))] \tag{18}$$

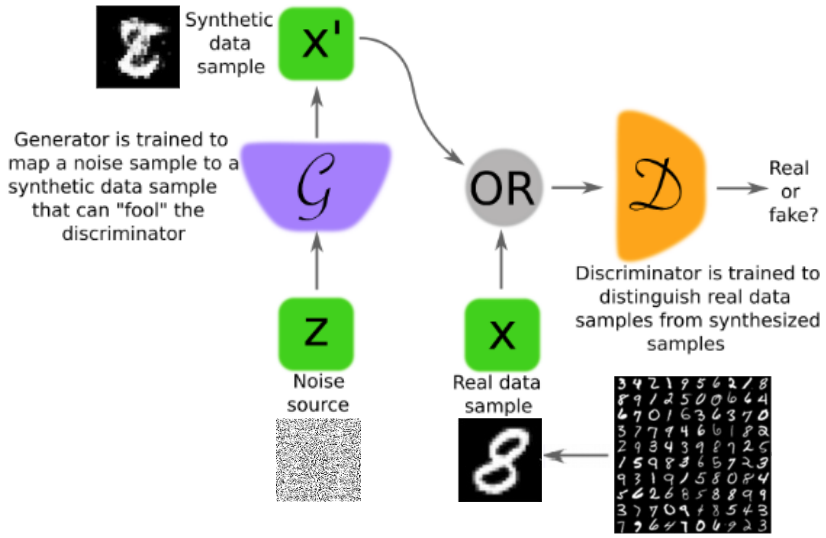

Figure 4. Schematic overview of a Generative Adversarial Network (GAN) [1] on MNIST data. There two networks, generator $\mathcal{G}$ and discriminator $\mathcal{D}$, are learned during optimization for a generative adversarial network. While the generator samples new images, the discriminator has to distinguish between generated images and the real dataset.

Question 2.1 (6 points)

a) (4pts) Explain the two terms in the GAN training objective defined in Equation 18 based on what part of the adversarial loss they represent. (*Two sentences per term are sufficient*)
b) (2pts) What is the value of $V(D, G)$ after training has converged? Explain it shortly.

Question 2.2 (4 points)

Early on during training, the $\log(1-D(G(Z)))$ term can be problematic for training the GAN. Explain why this is the case and how it can be solved.

2.2 Building a GAN

Now that the objective is specified and it is clear how the generator and discriminator should behave, we are ready to implement a GAN. In this part of the assignment you will implement a GAN in PyTorch. Training a GAN includes two steps per iteration. First, we randomly sample a latent vector $Z \in \mathbb{R}^{B \times d_z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ where B is the batch size and d_z the dimensionality of the latent space. This latent vector is forwarded through the generator to create $x' = G(Z)$, and we can use x' to calculate the loss. In the second step, you forward a fake batch $\tilde{x}$ and a "real" batch x (the batch from the dataset) through the discriminator. Based on the discriminator's output, you calculate the gradients for D. Note that $\tilde{x}$ and x' can be the same in a training step, but can also be two different samples from G. In the implementation, we will use the latter version as it is less error prone to implement (guidance is provided in the templates). It is also recommended to use the solution of Question 2.3 to stabilize training.

In the code directory `part2`, you find the templates to use for implementing the GAN. Similarly to the VAE implementation, we provide a training loop template in PyTorch Lightning (`train_pl.py`), and one in plain PyTorch (`train_torch.py`). **You only need to implement one of the two training loop templates**. You also need to implement the generator and discriminator in the file `models.py`. We specified a recommended architecture to start with, but you are allowed to experiment with your own ideas for the models (see e.g. GAN hacks for tips and tricks on training strong GANs). For the sake of the assignment, it is sufficient to use the recommended architecture to achieve full points. Use the provided unittests to ensure the correctness of your generator and discriminator implementation. Read the provided README to become familiar with the code template.

Question 2.3 (12 points)

Implement a GAN in the provided templates, and train it on the MNIST dataset. Provide a short description (no more than 10 lines) of the used architectures for generator and discriminator, and your training steps.

Additionally, sample 64 images (8×8 grid) from your trained GAN and include these in your report. Do this at the start of training, after 10 epochs, and after 250 epochs when training has terminated. Shortly describe the quality and differences over training.

Remark: We do not grade on a scale of who gets the best images, but you should be able to get decent predictions with the default model (i.e. in most generations of the trained model, you should be able to identify the digit that was generated, but they might not be perfect). Nevertheless, feel free to experiment with the architectures and hyperparameters.

Question 2.4 (4 points)

In GANs, we can interpolate between 2 images by interpolating their latent noise vector. For a trained model, sample 2 images from your GAN and interpolate between these two digits in latent space. Include 4 examples of those interpolations in your report here, and explain what you see. Use 5 interpolation steps, resulting in 7 images for each of the 4 examples (including start and end point).

Question 2.5 (4 points)

There are some problems during vanilla GANs optimization including but not limited to mode collapse, non-convergence and instability. Do you have any problems in your implementation? Shortly introduce one of your problems, explain how it happened and how you solved it. (*In case you have not experienced any, pick any of problems mentioned above, explain it and how it could be potentially solved.*)

3 Generative Normalizing Flows **(20 points)**

Time estimate: 3 hours (3 hours theory)

Thus far, two methods for high dimensional density estimation were presented: VAEs and GANs. However, there is another third type of generative models, which is getting more attention in literature recently: Flow-based generative models.

Similar to VAEs, they have a distribution over latent variables. Unlike VAEs, there is an exact correspondence (not a distribution) between a single data point and a latent representation. This is illustrated in Figure 5.

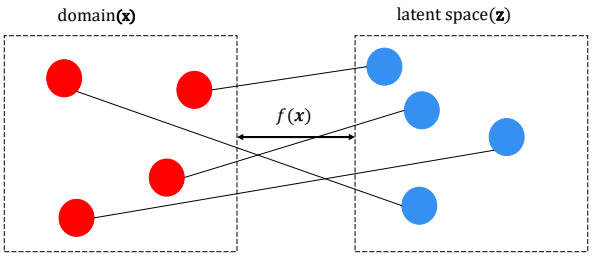

Figure 5. Flow-based generative models learn bijective mappings between high-dimensional input variable **y** and its latent representation **z**. Both sets of variables are connected by an invertible transformation function $f(\mathbf{x})$

.

Flow-based models utilize normalizing flows. Rezende et al. [4] defines it as follows: "A normalizing flow describes the transformation of a random variable through a sequence of invertible mappings. By repeatedly applying the rule for change of variables, the initial density 'flows' through the sequence of invertible mappings. At the end of this sequence we obtain a valid probability distribution and hence this type of flow is referred to as a normalizing flow."

The idea is very similar to VAEs in that every datapoint can be obtained by sampling a latent variable and then transforming it through a sequence of invertible mappings to get an actual data point in the end. If z is the latent variable and x the actual data point, a function f is invertible if $z = f(x)$ and $x = f^{-1}(z)$. Note that this requires a 1-to-1 correspondence, and doesn't work for functions like $f(x) = x^2$ where we cannot map the output back to a unique input. You can see a visualization of the invertible mapping of a flow-based models in Figure 5. To ensure that the distribution is valid at each transformation step (integrates to 1), Normalizing Flows are using the change of variables formula. The formula of change of variables allows us to express the distribution over x, $p_x(x)$, by a distribution in the latent space z, $p_z(z)$, and an invertible mapping f:

$$\int p_x(x)dx = \int p_z(z)dz = 1 \qquad \text{(by definition of a probability distribution)} \tag{19}$$

$$\Leftrightarrow p_x(x) = p_z(z)\left|\frac{dz}{dx}\right| = p_z(f(x))\left|\frac{df(x)}{dx}\right| \tag{20}$$

Note that this only works if the transformation f is **invertible** and **smooth**. In summary, a normalizing flow models a probability distribution $p_x(x)$ as follows:

$$z = f(x); \quad x = f^{-1}(z); \quad p_x(x) = p_z(z)\left|\frac{df(x)}{dx}\right| \tag{21}$$

where x is the variable in the output domain and z the latent variable. The factor $\left|\frac{df}{dx}\right|$ re-scales the transformed distribution to a valid density, as given by the change of variables formula.

Question 3.1 (5 points)

Assume you have a base density $z \sim$ Uniform(a,b) and a transformation function $f : X \to Z$ where z:

$$z = f(x) = x^3$$

Write down the probability density $p_x(x)$ in terms of x using the change of variables formula from equation 21. Check your answer by integrating the result you found, and verify that: $\int_{-\infty}^{\infty} p_x(x)dx = 1$.

Food for thought

How does the new probability density $p_x(x)$ look like? If you want, you can use tools like Desmos to visualize the distribution.

3.1 Change of variables for Neural Networks

In practice, Normalizing Flows consist of multiple invertible functions $f_1, f_2, ..., f_L$ that are applied in sequence: $z = f_L \circ f_{L-1} \circ ... \circ f_1(x)$. The formulas still hold for this case because applying multiple invertible functions in sequence is yet still invertible. When x is the data we want to model, we specify $p(z)$ to be a pre-defined distribution such as a multivariate Gaussian, and we optimize the log-likelihood $\log p_x(x)$. Specifically, let h_i denote the output of f_i with $h_0 = x$ and $h_L = z$, then the objective can be written as:

$$\log p_x(x) = \log p_z(z) + \sum_{l=1}^{L} \log\left|\frac{dh_l}{dh_{l-1}}\right| \tag{22}$$

Equations 21 and 22 describe the setting where we use a normalizing flow model to learn a univariate distribution. However, in all of the generative models we have looked at so far, we have used neural networks to transform *multivariate* random variables. Neural networks can be seen (ignoring the non-smooth part of some non-linearities) as smooth maps between spaces of real numbers. To use neural networks in normalizing flows models, we first need to understand what equations 21 and 22 would look like if $f : \mathbb{R}^m \to \mathbb{R}^m$ is an invertible smooth mapping and $x \in \mathbb{R}^m$ is a multivariate random variable. More formally, this translates to:

$$\mathbf{z} = f(\mathbf{x}); x = f^{-1}(\mathbf{z}); p_\mathbf{x}(\mathbf{x}) = p_\mathbf{z}(\mathbf{z})\left|\det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}\right| \tag{23}$$

$$\log p_\mathbf{x}(\mathbf{x}) = \log p_\mathbf{z}(\mathbf{z}) + \sum_{l=1}^{L} \log\left|\det \frac{\partial \mathbf{h_l}}{\partial \mathbf{h_{l-1}}}\right| \tag{24}$$

The difference compared to the univariate case is that we obtain a Jacobian matrix $\mathbf{J} = \frac{\partial \mathbf{h_l}}{\partial \mathbf{h_{l-1}}}$ instead of a single scalar. Therefore we take the determinant of the Jacobian matrix in

order to obtain a single scalar again. Looking at this equation, we have to think about possible limitations of Normalizing Flows on multivariate distributions.

Question 3.2 (6 points)

a) (3pts) What are the theoretical constraints that have to be set on the components h_l of the function f to make this equation computable? (Hint: think about the number of dimensions in h_l and h_{l-1})
b) (3pts) Even if we ensure the theoretical properties mentioned in the previous question, what might be an optimization issue using Equation 23? What might be a problem when using your neural network at test time to sample data points if you have limited computational resources?

3.2 Building a flow-based model

In the last sections, we dealt with the theory of general normalizing flows. Now that you understand the theory, it's time to look at what the practical implementation should look like. Let's start with discussing how to train a model.

Question 3.3 (9 points)

a) (4pts) The change-of-variables formula assumes continuous random variables. Imagine you want to train a flow for the task of image modeling, where images are usually stored as discrete integers. What would the learned distribution look like? How would you fix it? (Hint: Take a look at Tutorial 10).
b) (5pts) Within 10 lines, describe the steps you need to take to train a general flow-based model, evaluate it after training, and sample from it. In particular, explain the input and output of the model for each step.

3.3 Toy application: Transforming Gaussian densities with flows

If you want to gain some deeper understanding and practical experience with flow-based generative models, you can implement a flow model to transform a base density to a more complicated one on a simple toy example in the bonus question below. As base density for this question we define a bivariate gaussian with diagonal covariance matrix with PDF:

$$p(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left(-\frac{x^2}{2\sigma_x^2} - \frac{y^2}{2\sigma_y^2}\right) \tag{25}$$

Remember that if two variables X and Y are normally distributed, their sum also follows a normal distribution. That is if

$$\begin{aligned} x &\sim \mathcal{N}\left(\mu_x, \sigma_x^2\right) \\ y &\sim \mathcal{N}\left(\mu_y, \sigma_y^2\right) \\ z &= x + y \end{aligned} \tag{26}$$

then

$$z \sim \mathcal{N}\left(\mu_x + \mu_y, \sigma_x^2 + \sigma_y^2\right)$$

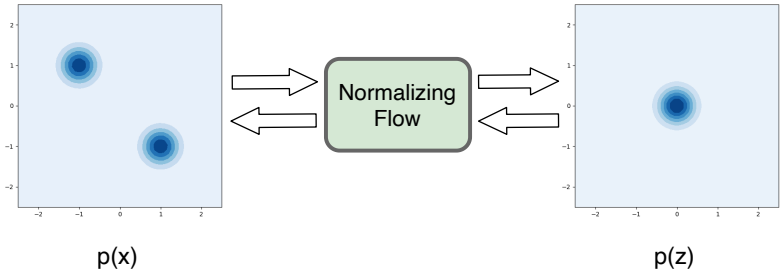

Figure 6. Visualization of the task at hand. We want to model the bimodal Gaussian $p(x)$ via a Normalizing Flow which transforms the input distribution to a unimodal normal distribution $p(z)$.

Question 3.4 (Bonus - 5 points)

Distributions. The goal is to use a generative flow based model to transform the base density described in equation 25 into a bimodal gaussian with diagonal covariance matrix (the target density). So first, complete the template for the base and target density in `part3/distributions.py`. You need to implement a logic for log-probability computation and the sampling procedure. Use the result on the sum of normally distributed random variables stated in 26 to obtain the log probability of the target density. You can check your implementation of the densities by plotting their contours with the visualization functions we provide in `part3/utils.py`.

Flow model. The flow model is build of coupling layers which utilizes boolean masks for mixing the input variables. Take a look here if you want to see how masking is used in the coupling layers to predict scale and translation parameters and further transform the inputs. The model class implementation is located in the `part3/model.py` file. Implement the missing code where prompted.

Training Loop. We have provided you with the training loop in the `part3/train.py` file. Implement the missing code and train your model for the default parameter settings we provided in the template. Include a plot of the bits per dimension curve over training steps in your report. *Hint: For reference, your model should achieve a bits per dimension value of about 0.84 after 40 epochs.*

Visualizations. Finally, visualize the learned density and compare it to the target and base densities you have already plotted. In `part3/utils.py`, you will find the visualization functions we provided. Include plots for all 3 densities in your report. What do you observe ? Explain what you see.

4 Conclusion **(5 points)**

Question 4.1 (5 points)

Write a short conclusion to your report in which you compare VAEs, GANs and Flow-based models. Comment on how the different models use a latent representation $\mathbf{z}$, and what advantage each model has compared to the others. Try to use no more than 15 lines for your conclusion. You do not need to go into every detail, but try to summarize the main points.

(Hint: Take a look at Figure 1 from this blogpost. You are allowed to include it in your answer.)

Report

We expect each student to answer all questions in this assignment in a report. Please clearly mark each answer by a heading indicating the question number. Again, use the NIPS LaTeX template as provided here: `https://nips.cc/Conferences/2018/PaperInformation/StyleFiles`.

Deliverables

On Canvas you will find **two submission hand-ins**. One hand-in for the code and one hand-in for the report. You need to submit all Python code and the report in a ZIP file and separately only the report as PDF in the dedicated submissions on Canvas.

For the code, please preserve the directory structure as provided in the Github repository for this assignment. Give each file the following name: `lastname_assignment3.zip` and `lastname_assignment3.pdf` where you insert your lastname. Please submit your deliverable through the dedicated submissions on Canvas. We cannot guarantee a grade for the assignment if the deliverables are not handed in according to these instructions.

The deadline for this assignment is December 20$^{\text{th}}$ at 23:59.

References

[1] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018. 10

[2] 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. 1

[3] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *International Conference on Learning Representations (ICLR)*, 2014. 1, 9

[4] 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. 1, 13