# An Introduction to Deep Generative Modeling

Lars Ruthotto[1] and Eldad Haber[2]

[1]Department of Mathematics, Emory University, Atlanta, GA, USA
[2]Department of Earth and Ocean Sciences, University of British Columbia, Vancouver, BC, Canada

April 13, 2021

**Abstract**

Deep generative models (DGM) are neural networks with many hidden layers trained to approximate complicated, high-dimensional probability distributions using a large number of samples. When trained successfully, we can use the DGMs to estimate the likelihood of each observation and to create new samples from the underlying distribution. Developing DGMs has become one of the most hotly researched fields in artificial intelligence in recent years. The literature on DGMs has become vast and is growing rapidly. Some advances have even reached the public sphere, for example, the recent successes in generating realistic-looking images, voices, or movies; so-called deep fakes. Despite these successes, several mathematical and practical issues limit the broader use of DGMs: given a specific dataset, it remains challenging to design and train a DGM and even more challenging to find out why a particular model is or is not effective. To help advance the theoretical understanding of DGMs, we introduce DGMs and provide a concise mathematical framework for modeling the three most popular approaches: normalizing flows (NF), variational autoencoders (VAE), and generative adversarial networks (GAN). We illustrate the advantages and disadvantages of these basic approaches using numerical experiments. Our goal is to enable and motivate the reader to contribute to this proliferating research area. Our presentation also emphasizes relations between generative modeling and optimal transport.

**Keywords:** Deep Generative Models, Machine Learning, Deep Learning, Optimal Transport, Normalizing Flow, Variational Autoencoder, Generative Adversarial Network

## 1 Motivation

Applications of deep generative models (DGM), such as creating fake portraits from celebrity images, have recently made headlines. The advent of these so-called deep fakes poses considerable societal and legal challenges, but also promise new beneficial technologies [10]. Those include new scientific applications of DGMs, for example, in physics and computational chemistry [9, 35, 7].

Fueled by these headlines and the potential applications across scientific disciplines, there has been an explosion in research activity in generative modeling in recent years. Due to the high volume and frequency of publications in this area, this article does not attempt to provide a comprehensive review. Instead, we seek to provide a mathematical introduction to the field, use an in-depth discussion of three main approaches to show the potential of DGMs, and expose open challenges. We also aim at illustrating similarities between generative modeling and other fields of applied mathematics, most importantly, optimal transport (OT) [14, 49, 39]. For a more comprehensive view of the field, we refer to the monographs on deep learning [18, 24], variational autoencoders (VAE) [29, 42, 30], and generative adversarial nets (GAN) [17]. To enable progress in this area, we provide the codes used to generate examples in this paper as well as interactive iPython notebooks in our Github repository at `https://github.com/EmoryMLIP/DeepGenerativeModelingIntro`.

Deep generative models are neural networks with many hidden layers trained to approximate complicated, high-dimensional probability distributions. In short, the ambitious goal in DGM training is to learn an unknown or intractable probability distribution from a typically small number of independent

and identically distributed samples. When trained successfully, we can use the DGM to estimate the likelihood of a given sample and to create new samples that are similar to samples from the unknown distribution. These problems have been at the core of probability and statistics for decades but remain computationally challenging to solve, particularly in high dimensions.

Despite many recent advances and success stories, several open challenges remain in the field of generative modeling. This paper focuses on explaining three key mathematical challenges.

1. DGM training is an ill-posed problem since uniquely identifying a probability distribution from a finite number of samples is impossible. Hence, the performance of the DGM will depend heavily on so-called hyperparameters, which include the design of the network, the choice of training objective, regularization, and training algorithms.
2. Training the generator requires a way to quantify its samples' similarity to those from the intractable distribution. In the approaches considered here, this either requires the inversion of the generator or comparing the distribution of generated samples to the given dataset. Both of these avenues have their distinct challenges. Inverting the generator is complicated in most cases, particularly when it is modeled by a neural network that is nonlinear by design. Quantifying the similarity of two probability distributions from samples leads to two-sample test problems, which are especially difficult without prior assumptions on the distributions.
3. Most common approaches for training DGMs assume that we can approximate the intractable distribution by transforming a known and much simpler probability distribution (for instance, a Gaussian) in a latent space of known dimension. In most practical cases, determining the latent space dimension is impossible and is left as a hyperparameter that the user needs to choose. This choice is both difficult and important. With an overly conservative estimate, the generator may not approximate the data well enough, and an overestimate can render the generator non-injective, which complicates the training.

To increase the accessibility of the paper, we will keep the discussion as informal as possible and sacrifice generality for clarity where needed.

The remainder of the paper is organized as follows. In Section 2 we describe the generative modeling problem mathematically. In Section 3, we present finite [41, 12] and continuous normalizing flows [20, 51, 50, 15, 36]. In Section 4, we introduce variational autoencoders [29, 42, 30]. In Section 5, we introduce Generative Adversarial Networks [19, 3]. In Section 6, we provide a detailed discussion and comparison of the three approaches. In Section 7, we conclude the paper and highlight a few important directions of future research.

# 2 Mathematical Formulation and Examples

This section establishes our notation, defines and illustrates the deep generative modeling problem, presents two numerical examples used to demonstrate the different approaches, and provides a high-level overview of the DGM training problem.

## 2.1 General Set Up

The key goal in generative modeling is to learn a representation of an intractable probability distribution $\mathcal{X}$ defined over $\mathbb{R}^n$, where $n$ typically is large, and the distribution is complicated; consider, for example, a multimodal distribution with disjoint support. To this end, we can use a potentially large, but typically finite, number of independent and identically distributed (i.i.d) samples from $\mathcal{X}$ that we refer to as the training data. Unlike standard statistical inference where a mathematical expression for the probability is sought, the goal is to obtain a generator

$$g : \mathbb{R}^q \to \mathbb{R}^n \tag{1}$$

that maps samples from a tractable distribution $\mathcal{Z}$ supported in $\mathbb{R}^q$ to points in $\mathbb{R}^n$ that resemble the given data. In other words, we assume that for each sample $\mathbf{x} \sim \mathcal{X}$ there is at least one point $\mathbf{z} \sim \mathcal{Z}$, such that $g(\mathbf{z}) \approx \mathbf{x}$. We denote the transformation of the latent distribution $\mathcal{Z}$ as $g(\mathcal{Z})$. Having a generator that can map points from the simple distribution, $\mathcal{Z}$, to the intractable distribution, $\mathcal{X}$, allows us to generate samples from the complicated space $\mathcal{X}$, which is desired in many applications.

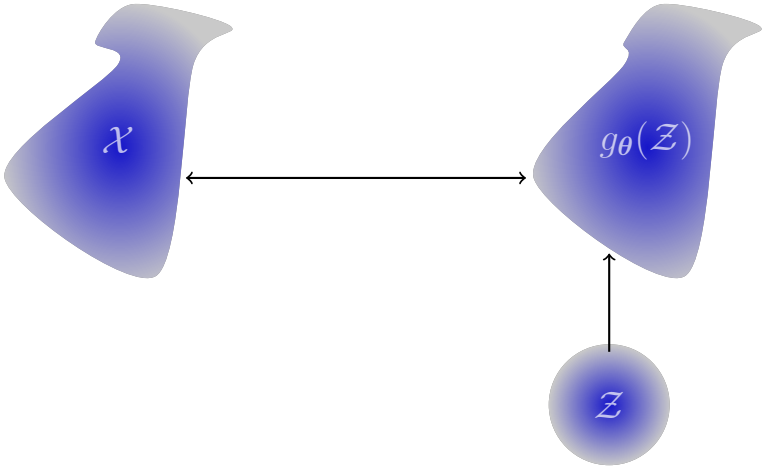


Figure 1: A deep generative model, $g_{\boldsymbol{\theta}}$, is trained to map samples from a simple distribution, $\mathcal{Z}$, (bottom right) to the more complicated distribution $g_{\boldsymbol{\theta}}(\mathcal{Z})$ (top right), which is similar to the true distribution $\mathcal{X}$ (top left). Finding an objective function that quantifies the discrepancy between the generated samples and the original examples is the key obstacle to training generative models. This is particularly difficult in the absence of point-to-point correspondences between data samples and latent variables.

Since the vector $\mathbf{z}$ that results in a given vector $\mathbf{x}$ is generally unknown, it is common to refer to it as the latent variable and call $\mathcal{Z}$ the latent space. As is common, we will assume that $\mathcal{Z}$ is a univariate Gaussian in $\mathbb{R}^p$. This is without loss of generality and in principle, $\mathcal{Z}$ can be any tractable distribution; that is, we require the ability to sample from $\mathcal{Z}$ and, in some cases, compute the probability $p_{\mathcal{Z}}(\mathbf{z})$. We illustrate our notation in Figure 1.

It is important to note that the latent space dimension, $q$, will generally be different from the dimension of the data space, $n$. For example, high-resolution images with millions of pixels do not really "live" in such a high-dimensional space since their content mostly gets preserved when reducing the resolution. Instead, there is a hidden manifold of typically unknown dimension in which the images reside. This further complicates the problem, and we discuss this point later in the paper.

Assuming the generator $g$ is known, we can generate new data points by sampling $\mathbf{z} \sim \mathcal{Z}$ and computing $g(\mathbf{z})$. In many applications, ranging from deep fakes to Bayesian statistics, generating new samples is the only goal. In addition, the generator can also be used to compute the likelihood or evidence of a particular sample $\mathbf{x}$ using marginalization

$$p_{\mathcal{X}}(\mathbf{x}) = \int p_g(\mathbf{x}|\mathbf{z}) p_{\mathcal{Z}}(\mathbf{z}) d\mathbf{z}, \tag{2}$$

where the likelihood $p_g(\mathbf{x}|\mathbf{z})$ measures how close $g(\mathbf{z})$ is to $\mathbf{x}$. Note that the exact computation of (2) is, in general, intractable due to the high-dimensionality of the integral. The choice of the likelihood function depends on the properties of the data. For real-valued data, it is common to use a Gaussian which leads to

$$p_g(\mathbf{x}|\mathbf{z}) = (2\pi\sigma)^{-\frac{n}{2}} \exp\left(-\frac{1}{2\sigma}\|g(\mathbf{z}) - \mathbf{x}\|^2\right), \tag{3}$$

where the choice of $\sigma > 0$ controls how narrow the likelihood is around the samples. For binary data, one typically assumes a Bernoulli distribution, which leads to

$$p_g(\mathbf{x}|\mathbf{z}) = \prod_{i=1}^{n} g(\mathbf{z})_i^{\mathbf{x}_i} \ \left(1 - g(\mathbf{z})_i\right)^{(1-\mathbf{x}_i)}. \tag{4}$$

Deriving $g$ from first principles is impossible or infeasible for most data sets of interest. For example, it may be challenging to model the process that transforms a sample from a univariate Gaussian to

an image of a celebrity. Therefore, it has become common in recent years to use generic function approximators such as neural networks with many hidden layers. This is the fundamental design concept in deep generative models (DGM), where $g$ is modeled using a feed-forward deep neural network (DNN). Advantages of DNNs include their ability to approximate functions in high dimensions effectively. We denote the DNN generator by $g_{\boldsymbol{\theta}}$ and its weights by $\boldsymbol{\theta} \in \mathbb{R}^{N_{\boldsymbol{\theta}}}$.

Defining the DNN architecture that defines $g_{\boldsymbol{\theta}}$, that is, choosing the number of layers and the operations involved in each layer, is a topic in its own right that we will not discuss in detail here. Instead, we will review a few examples from the literature in our numerical experiments below and refer the reader to the excellent introduction [24] and the comprehensive textbook [18] for in-depth discussions and more options. Our choice is made for conciseness and should not divert from the fact that choosing an effective architecture is critical and complicated by the lack of theoretical guidelines. For example, the quality of the architecture impacts our ability to model the generative process and our ability to solve the learning problem, that is, to train the parameters of the generator.

## 2.2 Testbed Examples

We use two examples to illustrate and compare the different approaches to deep generative modeling. Our goal is not to improve the state-of-the-art on those common benchmark problems but to keep the models and their implementation as simple as possible and closely match the presented derivations. We encourage the reader to look under the hood and perform more in-depth experiments and provide our implementation at `https://github.com/EmoryMLIP/DeepGenerativeModelingIntro`.

**Example 1 (Moons)** *We use a two-dimensional example to help visualize the deep generative modeling problem, the data and latent distributions, and the intermediate steps of the generation process. Here, we consider the* `moons` *example from the* `scikit-learn` *package [38]. The implementation provides an infinite number of (pseudo) random samples from a complicated distribution whose support is split into two disjoint regions of equal mass shaped like half-moons; see Figure 2 for a visualization. The user can control the width of the half-moon shapes. For the setting used in our plot, the width is sufficiently wide such that the support of the underlying distribution has non-zero volume in $\mathbb{R}^2$. Hence, we use a $q = 2$ dimensional latent space and seek to find a generator that maps the standard normal distribution to the data distribution. It is important to note that if we reduced the width of the moon shapes considerably, their intrinsic dimension would reduce to one, and we would expect this approach to fail or at least provide suboptimal results. Even though the optimal generator is discontinuous, one common approach to generative modeling where $q = n$ is to restrict the search to smooth and invertible models; see Section 3. This modeling choice allows us to efficiently compute and optimize each sample's likelihood since we can determine the latent variable associated with each sample. Still, we expect large derivatives of the generator in parts of the latent space as we try to transform a uni-modal Gaussian to a bi-modal distribution; for more insight on this issue and ways to stabilize the generator using a mixture model as latent distribution, see [23]. As we will see, upon suitable construction of the generator, this results in a relatively straightforward training problem compared to the more advanced methods needed when $q \neq n$.*

**Example 2 (MNIST)** *As a high-dimensional example in which the data's intrinsic dimensionality is clearly less than $n$, we consider the well-studied MNIST dataset [31]; see Figure 3. The dataset consists of gray-valued digital images, each having $28 \times 28$ pixels and showing one hand-written digit. The dataset provides a finite number of images that are divided into 60,000 training and 10,000 test images. To train the generator, we do not require labels; however, we demonstrate in Figure 8 that the embedding of the data points into the latent space in this example roughly clusters the samples based on the digit shown.*

*The first obstacle to setting up the DGM training is that the intrinsic dimension of the MNIST dataset is unknown, which renders choosing the dimension of the latent space non-trivial. While each image contains $n = 784$ pixels, the support of $\mathcal{X}$ will likely lie in a subset of a much lower dimension. Also, since the images are grouped into ten different classes, one can expect the support to be disjoint with a substantial distance between the different clusters.*

*Despite these conceptual challenges, we will demonstrate that DGMs can be trained effectively and at modest computational costs to create realistic-looking images. In our example, we seek to train a DGM such that it maps samples from the two-dimensional standard normal distribution to realistic images.*

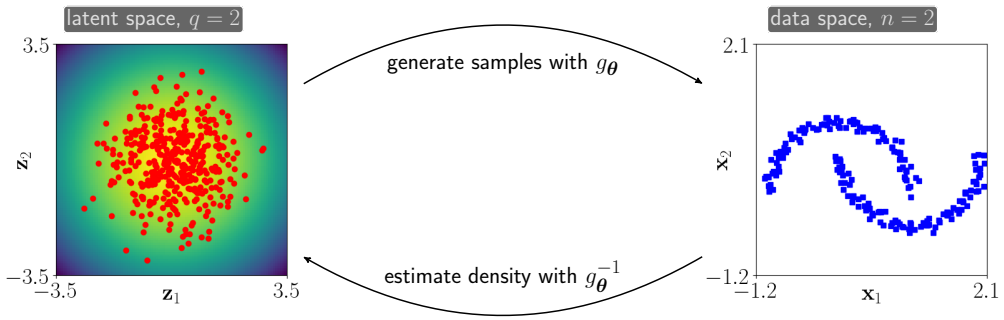


Figure 2: Visualizing the training data of the two moons dataset in Example 1. Here, the data distribution (samples represented by blue squares in right subplot) form two disjoint half moon shaped clusters. As the dataset does not live on a lower dimensional manifold, we assume an intrinsic dimension of two and try to find a generator from the $q = 2$ dimensional normal distribution (left subplot) to the data. In practice, it is common to model the generator as an invertible transformation, which simplifies the training; see Section 3.

*This choice is made so that we can easily visualize the latent space; however, we note that using a larger latent space dimension may improve the quality of the generated images. Similar to [40], we define the generator as a three-layer convolutional neural network (CNN) that transforms an input sample* $\mathbf{z} \in \mathbb{R}^2$ *to a vectorized image* $g_{\boldsymbol{\theta}}(\mathbf{z}) \in \mathbb{R}^{784}$ *using the following three steps*

$$\begin{aligned} \mathbf{w}^{(1)} &= \sigma_{\mathrm{ReLU}}\left(\mathcal{N}\left(\mathbf{K}^{(1)}\mathbf{z} + \mathbf{b}^{(1)}\right)\right), \\ \mathbf{w}^{(2)} &= \sigma_{\mathrm{ReLU}}\left(\mathcal{N}\left(\mathbf{K}^{(2)}\mathbf{w}^{(1)} + \mathbf{b}^{(2)}\right)\right), \\ g_{\boldsymbol{\theta}}(\mathbf{z}) &= \sigma_{\mathrm{sigm}}\left(\mathbf{K}^{(3)}\mathbf{w}^{(2)} + \beta\right). \end{aligned} \tag{5}$$

*Here,* $\mathbf{K}^{(1)}, \mathbf{K}^{(2)}, \mathbf{K}^{(3)}$ *are linear operators,* $\mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \beta$ *are bias terms,* $\mathcal{N}$ *denotes a batch normalization layer,* $\sigma_{\mathrm{ReLU}}(x) = \max\{x, 0\}$ *is the rectified linear unit, and* $\sigma_{\mathrm{sigm}}(x) = (1 + \exp(-x))^{-1}$ *is the sigmoid function. The activation functions,* $\sigma_{\mathrm{ReLU}}$ *and* $\sigma_{\mathrm{sigm}}$*, are applied element-wise. For ease of notation, we collectively denote the parameters of the model as* $\boldsymbol{\theta}$*; that is,* $\boldsymbol{\theta}$ *is a vector containing the parameters of* $\mathbf{K}^{(1)}, \mathbf{K}^{(2)}, \mathbf{K}^{(3)}, \mathbf{b}^{(1)}, \mathbf{b}^{(2)}, \beta$*.*

*We now describe our generative model in more detail. The matrix* $\mathbf{K}^{(1)}$ *in the first layer is of size* $(64 \cdot 7 \cdot 7) \times 2$*, that is, it transforms the input* $\mathbf{z} \in \mathbb{R}^2$ *into 64 images of size* $7 \times 7$*, also called channels. We add the bias vector* $\mathbf{b}^{(1)} \in \mathbb{R}^{64}$ *channel-wise and apply batch normalization [25] the output of the affine transformation before, finally, using the activation function. The matrix* $\mathbf{K}^{(2)}$ *in the second layer is of size* $(32 \cdot 14 \cdot 14) \times (64 \cdot 7 \cdot 7)$*. It maps the input images* $\mathbf{w}^{(1)}$ *to 32 images with* $14 \times 14$ *pixels each by computing the transpose of strided convolutions whose stencils are* $4 \times 4$*. As in the first layer, the bias* $\mathbf{b}^{(2)} \in \mathbb{R}^{32}$ *shifts each channel separately. In the final layer, the matrix* $\mathbf{K}^{(3)}$ *is of size* $784 \times (32 \cdot 14 \cdot 14)$*; that is, it provides a single image with* $28 \times 28$ *pixels by computing a linear combination of transposed, strided convolutions applied to all the channels. The bias,* $\beta$ *is a scalar. Due to the sigmoid activation function, the entries of* $g_{\boldsymbol{\theta}}(\mathbf{z})$ *are all between zero and one.*

*Overall, the generator has* $N_{\boldsymbol{\theta}} = 42,913$ *trainable weights that we initialize randomly and then train by minimizing an objective function. The specific construction of the objective function is the main difference between the approaches. Note that we cannot assume the generator to admit an inverse, unlike in the two-dimensional example.*

## 2.3 Training the Generator: A High-level Overview

In the remainder of the paper, we will discuss three main approaches for training the DGM $g_{\boldsymbol{\theta}}$ using samples from $\mathcal{X}$. Their common goal is to learn a parameter $\boldsymbol{\theta}$ such that new samples, $g_{\boldsymbol{\theta}}(\mathbf{z})$ where $\mathbf{z} \sim \mathcal{Z}$, are statistically indistinguishable from samples from the training data. In other words, we train

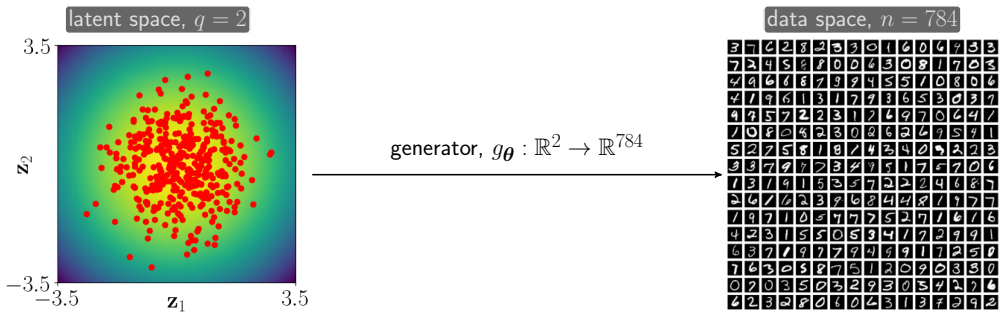


Figure 3: Illustration of the MNIST image generation process in Example 2. Here, the intrinsic dimension of the dataset (right) is unknown but assumed to be much less than the number of pixels per image, $n = 784$. In our example, we define the latent variable to be distributed according to the $q = 2$-dimensional standard normal distribution (left). Compared to the moons example (see Figure 2), the generator cannot be assumed to be invertible. This complicates the density estimation and the training process.

$\boldsymbol{\theta}$ so that $g_{\boldsymbol{\theta}}$ transforms the latent probability distribution, $\mathcal{Z}$, to the probability distribution of the data, $\mathcal{X}$. Determining the distance between two distributions is a two-sample hypothesis test problem, which is very difficult, especially for complicated distributions in high dimensions. Therefore, we will also judge the quality of the generator visually.

We will see correspondences between the latent variables and the data samples can help avoid the two-sample test problem. One key obstacle to this goal is the unsupervised nature of the problem; that is, we do not have pairs $(\mathbf{x}, \mathbf{z})$ of corresponding samples from the data and latent distribution, respectively. Instead, we have many samples from the data space, $\mathcal{X}$, and the function, $g_{\boldsymbol{\theta}}$ that we can use to create new samples. Therefore, if we want to avoid the challenging two-sample problem, we need to establish correspondences, for example, by inverting $g_{\boldsymbol{\theta}}$ or using statistical inference techniques.

We seek to exemplify these challenges using numerical experiments with the examples from Section 2.2. The three state-of-the-art approaches, which are built on different assumptions on the data, modeling choices, and numerical techniques, are:

- In Sec. 3, we describe two ways to construct an invertible DGM $g_{\boldsymbol{\theta}}$ and apply them to the moons dataset from Example 1. In the first one, we concatenate a finite number of invertible functions. In the second one, we model $g_{\boldsymbol{\theta}}$ as a trainable dynamical system. These approaches have become known as normalizing flows and continuous normalizing flows, respectively. When $g_{\boldsymbol{\theta}}$ and its inverse are continuously differentiable, we can avoid the integral in (2) and compute the likelihood of a sample using the change of variables formula. Hence, this assumption simplifies the training and also enables links to optimal transport (OT). The invertibility of the model assumes that $q = n$, which is limiting in many real-world datasets. Still, we can use (continuous) normalizing flows as building blocks in more powerful approaches; for example, we try to reduce the space dimension and then use the flow in the latent space.
- In Sec. 4, we discuss variational autoencoders (VAEs) [30] that use a probabilistic model to establish relations between the latent variables and data samples for non-invertible generators $g_{\boldsymbol{\theta}}$. VAEs are broadly applicable, for instance, in the realistic case where $q \ll n$. The key component in VAEs is a second neural network that approximates the intractable posterior density $p_{g_{\boldsymbol{\theta}}}(\mathbf{z}|\mathbf{x})$, which we will denote by $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ in the following. We use this distribution to derive the objective function required to train the generator. In our case, the objective function turns out to be a lower bound of the likelihood. VAE training then consists of minimizing the objective with respect to the weights of the generator and the approximate posterior simultaneously. A challenge in training VAEs is to maximize the overlap between the approximate posteriors and the latent distributions while minimizing the reconstruction loss.
- In Sec. 5, we consider the framework of generative adversarial networks (GAN) that, instead of attempting to invert the generator, tackles the two-sample test problem in the data space. Hence, GANs neither infer the latent variable nor compute approximate likelihoods. One way to distinguish

GAN approaches is the distance used to compare the actual distribution, represented by the samples, and the distribution implied by the generator, $g_{\boldsymbol{\theta}}$. We parameterize this distance using a second neural network called the discriminator. There exist several choices for this network, and we will discuss the classical approach based on a binary classifier and a common transport-based approach using a Wasserstein distance.

Before we dive into more details, it is important to note some of the similarities and differences. Normalizing flows apply only to the small set of problems in which the latent space dimension equals the intrinsic dimension of the data. Nonetheless, when these assumptions are justified, normalizing flows map $\mathcal{X}$ to $\mathcal{Z}$ and vice versa and allow direct estimation of the distances between $g_{\boldsymbol{\theta}}(\mathcal{Z})$ and $\mathcal{X}$ and between $g_{\boldsymbol{\theta}}^{-1}(\mathcal{X})$ and $\mathcal{Z}$. VAEs have less restrictive assumptions since they use a probabilistic model to infer the latent variable. However, this inference problem is not straightforward, given the nonlinearity of the generator. It is also non-trivial to ensure that the samples from the approximate posterior overlap sufficiently well with the latent distribution. Finally, GANs skip the challenges associated with estimating the latent variable and sample immediately from the latent distribution $\mathcal{Z}$. However, since we have no correspondence between the generated samples and the data points, quantifying the similarity between $g(\mathcal{Z})$ and $\mathcal{X}$ is highly non-trivial.

# 3 Finite and Continuous Normalizing Flows

The key idea in normalizing flows is to model the generator, $g_{\boldsymbol{\theta}}$, as a diffeomorphic and orientation-preserving function.[1] To this end, normalizing flow models assume that the latent space dimension, $q$, is equal to the dimension of the data space, $n$. While this is a significant restriction in practice, normalizing flows can be used as an add-on in other approaches that overcome this restriction. Under these assumptions, we can use the change of variables formula and approximate the likelihood of a given data point $\mathbf{x}$ in (2) by

$$\begin{aligned} p_{\mathcal{X}}(\mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{x}) &= p_{\mathcal{Z}}\left(g_{\boldsymbol{\theta}}^{-1}(\mathbf{x})\right) \det \nabla g_{\boldsymbol{\theta}}^{-1}(\mathbf{x}) \\ &= (2\pi)^{-\frac{n}{2}} \exp\left(-\frac{1}{2}\|g_{\boldsymbol{\theta}}^{-1}(\mathbf{x})\|^2\right) \det \nabla g_{\boldsymbol{\theta}}^{-1}(\mathbf{x}). \end{aligned} \tag{6}$$

In contrast to (2), no integration is needed, and we can evaluate $p_{\boldsymbol{\theta}}$ exactly when $\mathcal{Z}$ has a sufficiently smooth density, and we can efficiently compute both $g_{\boldsymbol{\theta}}^{-1}$ and its Jacobian determinant. In order to sample from $p_{\boldsymbol{\theta}}$, we further require an efficient way of evaluating $g_{\boldsymbol{\theta}}(\mathbf{z})$ to push forward samples from the latent distribution $\mathcal{Z}$. These requirements inform our modeling choices of the generator.

## 3.1 Maximum Likelihood Training

Given the parametric model for the generator, $g_{\boldsymbol{\theta}}$, we need to pick $\boldsymbol{\theta}$ so that we approximate the true likelihood function well, that is, that ideally equality holds in (6). One way to compare the two densities is to minimize the Kullback-Leibler (KL) divergence between $p$ and $p_{\boldsymbol{\theta}}$

$$\mathrm{KL}(p_{\mathcal{X}}||p_{\boldsymbol{\theta}}) = \int p_{\mathcal{X}}(\mathbf{x}) \log \frac{p_{\mathcal{X}}(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})} d\mathbf{x} = \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[\log\left(\frac{p_{\mathcal{X}}(\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{x})}\right)\right]. \tag{7}$$

While computing the KL divergence requires the unknown likelihood $p_{\mathcal{X}}(\mathbf{x})$ and is thus intractable, minimization with respect to $\boldsymbol{\theta}$ only requires samples from $\mathcal{X}$. Note that the KL divergence has a unique minimizer when $p_{\mathcal{X}} = p_{\boldsymbol{\theta}}$ but is not symmetric, that is, $\mathrm{KL}(p_{\mathcal{X}}||p_{\boldsymbol{\theta}}) \neq \mathrm{KL}(p_{\boldsymbol{\theta}}||p_{\mathcal{X}})$. Also, note that optimizing $\mathrm{KL}(p_{\boldsymbol{\theta}}||p_{\mathcal{X}})$ is intractable as it would require the true density.

Assuming $g_{\boldsymbol{\theta}}^{-1}$ is available, we seek to maximize the likelihood samples from $\mathcal{X}$ under $p_{\boldsymbol{\theta}}$, which is known as *maximum likelihood training*. Practically, we choose to minimize the negative log-likelihood

$$J_{\mathrm{ML}}(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[-\log p_{\boldsymbol{\theta}}(\mathbf{x})\right] \approx \frac{1}{s}\sum_{i=1}^{s}\left(\frac{1}{2}\left\|g_{\boldsymbol{\theta}}^{-1}\left(\mathbf{x}^{(i)}\right)\right\|^2 - \log\det\nabla g_{\boldsymbol{\theta}}^{-1}\left(\mathbf{x}^{(i)}\right) + \frac{n}{2}\log(2\pi)\right), \tag{8}$$

[1] A function $g : \mathbb{R}^n \to \mathbb{R}^n$ is diffeomorphic if it is invertible and both $g$ and $g^{-1}$ are continuously differentiable. If $g$ is also orientation-preserving, $\det \nabla g(z) > 0$.

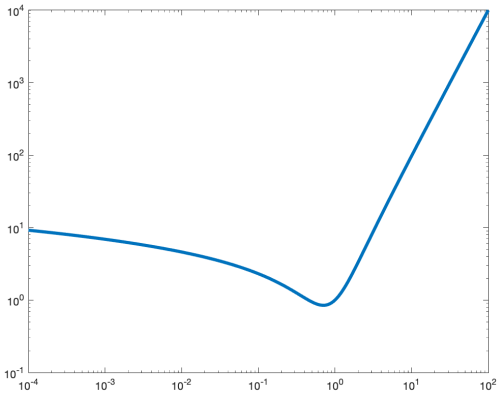


Figure 4: The objective function for a simple affine function $g_{\boldsymbol{\theta}}^{-1}(x) = \boldsymbol{\theta} x$ for a single example.

where $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(s)}$ are i.i.d. samples from $\mathcal{X}$ and $s$ is also called the batch size. We also note that $\mathrm{KL}(p_{\mathcal{X}} || p_{\boldsymbol{\theta}}) = J_{\mathrm{ML}}(\boldsymbol{\theta}) + \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[-\log p_{\mathcal{X}}(\mathbf{x})\right]$ and since the second term is constant with respect to $\boldsymbol{\theta}$, minimizing $J_{\mathrm{ML}}(\boldsymbol{\theta})$ is equivalent to minimizing the Kullback-Leibler divergence in (7).

The terms in Equation (8) can be also interpreted by a more classical regularized approach. The first term is minimized when $g_{\boldsymbol{\theta}}^{-1}(\mathbf{x}) = 0$ irrespectably of $\mathbf{x}$, that is, it prefers transformations that shrink the space. The second term can be viewed as mapping the volume around $\mathbf{x}$ and it has the exact opposite effect. It prefers transformations that are volume preserving. A plot of the function for a transformation in a single dimension is plotted in Figure 4. The approximation in (8) comes through the sampling of the distribution in $\mathcal{X}$ it is important to understand that the quality of the minimizer is only as good as the quality of this approximation, which for complicated, high-dimensional distributions can be poor.

The asymmetry of the KL divergence has important implications for the generators resulting from maximum likelihood training; see also [2]. On the one hand, the objective function will assign large values for points $\mathbf{x}$ at which the actual likelihood exceeds that implied by the generator; that is, $p_{\mathcal{X}}(\mathbf{x}) \gg p_{\boldsymbol{\theta}}(\mathbf{x})$. Hence, when trained well, the support of the density of the generator should cover the samples. On the other hand, the generator may produce samples with small likelihood since the KL divergence becomes small when $p_{\mathcal{X}}(\mathbf{x}) \ll p_{\boldsymbol{\theta}}(\mathbf{x})$.

To approximately minimize $J_{\mathrm{ML}}$, we use stochastic approximation methods such as stochastic gradient descent (SGD) and its variants; see the excellent survey [6]. In short, their iterations minimize the objective using gradients that are estimated from the current minibatch, which is re-sampled at every step. By avoiding using the entire dataset, SGD type methods seek to save computational costs and, empirically, often provide meaningful neural network models.

Let us now turn to the question of how to design the function $g_{\boldsymbol{\theta}}$. We assumed that it is a reversible diffeomorphic function that preserves the orientation of the data. In the following, we discuss how to build such a generator.

## 3.2 Finite Normalizing Flows

A finite normalizing flow [41, 12] is constructed by concatenating diffeomorphic transformations with tractable Jacobian determinants, which leads to the generator

$$g_{\boldsymbol{\theta}}(\mathbf{z}) = f_K \circ f_{K-1} \circ \cdots \circ f_1(\mathbf{z}). \tag{9}$$

In deep learning it is common to call $f_j$ the layers of the network and $K$ the depth of the network. Assuming that efficient expressions for the inverses of the layer functions $f_j$ are available, we can compute the maximum likelihood loss (8) using

$$g_{\boldsymbol{\theta}}^{-1}(\mathbf{x}) = f_1^{-1} \circ f_2^{-1} \circ \cdots \circ f_K^{-1}(\mathbf{x}) \quad \text{and} \quad \log\det \nabla g_{\boldsymbol{\theta}}^{-1}(\mathbf{x}) = \sum_{j=K}^{1} \log\det \nabla f_j^{-1}\left(\mathbf{y}^{(j)}\right). \tag{10}$$

Here, $\mathbf{y}^{(K)}, \mathbf{y}^{(K-1)}, \ldots, \mathbf{y}^{(1)}$ are the hidden features, $\mathbf{y}^{(0)} = \mathbf{z} = g_{\boldsymbol{\theta}}^{-1}(\mathbf{x})$, and we have

$$\mathbf{y}^{(j-1)} = f_j^{-1}\left(\mathbf{y}^{(j)}\right), \quad \text{for } j = K, K-1, \ldots, 1, \quad \text{with} \quad \mathbf{y}^{(K)} = \mathbf{x}.$$

Note that we can perform maximum likelihood training as long as we can compute the inverse of the generator and the log-determinant of its Jacobian. However, efficient sampling, which is our goal, requires efficient forward calculations as well. The key trade-off in normalizing flows is designing the layers $f_j$ to be expressive while also leading to tractable Jacobian determinants, and ideally same cost for evaluating $f_j$ and its inverse. These considerations allow us to group existing approaches by their ability to compute $g_{\boldsymbol{\theta}}$, $g_{\boldsymbol{\theta}}^{-1}$, or both:

- Examples of normalizing flows that can evaluate both the generator and its inverse efficiently are non-linear independent components estimation (NICE) [11] and real non-volume preserving (real NVP) flow [12], which we present in more detail below. A key idea in these approaches is that the layers partition the variables into two blocks and use components that are easy to invert. Both of these approaches belong to the more general class of invertible neural networks; see, e.g., the excellent literature review and application to inverse problems in [1].
- Examples of normalizing flows that can compute $g_{\boldsymbol{\theta}}$ efficiently but not its inverse include the planar and radial flows [41] and inverse autoregressive flows [28]. These approaches lack a closed-form expression for the inverse, which is needed to train the generator using the maximum likelihood objective function. Instead, they are commonly used in variational autoencoders, which we will discuss next.
- An example of a normalizing flow that can compute $g_{\boldsymbol{\theta}}^{-1}$ efficiently but not the generator is the masked autoregressive flow [37]. While these models can be trained straightforwardly using maximum likelihood training and provide efficient density estimates, their use to produce new samples is limited.

**Numerical Example: Real NVP For Moons Dataset** We apply the real-valued non-volume preserving (real NVP) flow [12] to our moons problem presented in Example 1. Our architecture and implementation is adapted from the excellent tutorial [4]. The $j$th layer splits its input $\mathbf{y}^{(j)} \in \mathbb{R}^2$ into its components $\mathbf{y}_1^{(j)}$ and $\mathbf{y}_2^{(j)}$. When $j$ is an even number, $f_j$ keeps the first component unchanged and transforms the second component using an affine transformation parameterized by $y_1^{(j)}$, that is,

$$f_j\left(\mathbf{y}^{(j)}\right) = \begin{bmatrix} \mathbf{y}_1^{(j)} \\ \mathbf{y}_2^{(j)} \cdot \exp\left(s_j\left(\mathbf{y}_1^{(j)}\right)\right) + t_j\left(\mathbf{y}_1^{(j)}\right) \end{bmatrix}, \tag{11}$$

where $s_j, t_j : \mathbb{R} \to \mathbb{R}$ are neural networks that model scaling and translation, respectively. We collect the trainable weights from all the layers in $\boldsymbol{\theta}$. The Jacobian of the $j$th layer reads

$$\nabla_{\mathbf{y}} f_j^{\top}\left(\mathbf{y}^{(j)}\right) = \begin{bmatrix} 1 & 0 \\ \mathbf{y}_2^{(j)} \cdot \exp\left(s_j\left(\mathbf{y}_1^{(j)}\right)\right) s_j'\left(\mathbf{y}_1^{(j)}\right) + t_j'\left(\mathbf{y}_1^{(j)}\right) & \exp\left(s_j\left(\mathbf{y}_1^{(j)}\right)\right) \end{bmatrix},$$

which simplifies the Jacobian determinant to

$$\det \nabla f_j\left(\mathbf{y}^{(j)}\right) = \exp\left(s_j\left(\mathbf{y}^{(j)}\right)\right). \tag{12}$$

We also note that, independent of the specific choice of the networks $s_j$ and $t_j$, the inverse of the layer is

$$f_j^{-1}\left(\mathbf{y}^{(j)}\right) = \begin{bmatrix} \mathbf{y}_1^{(j)} \\ \left(\mathbf{y}_2^{(j)} - t_j\left(\mathbf{y}_1^{(j)}\right)\right) \odot \exp\left(-s_j\left(\mathbf{y}_1^{(j)}\right)\right) \end{bmatrix}. \tag{13}$$

When $j$ is an odd number, the roles of the components of $\mathbf{y}$ are interchanged. We note that this leads to a well-conditioned flow as long as the magnitude of the scaling remains bounded to a small number.

In our example, we use a normalizing flow with $K = 6$ real NVP layers. For every $j = 1, 2, \ldots, 6$ the neural networks that parameterize the scaling and translation of the layers ($s_j$ and $t_j$, respectively) each consist of three affine layers with a hidden dimension of 128. The first two layers use a leaky

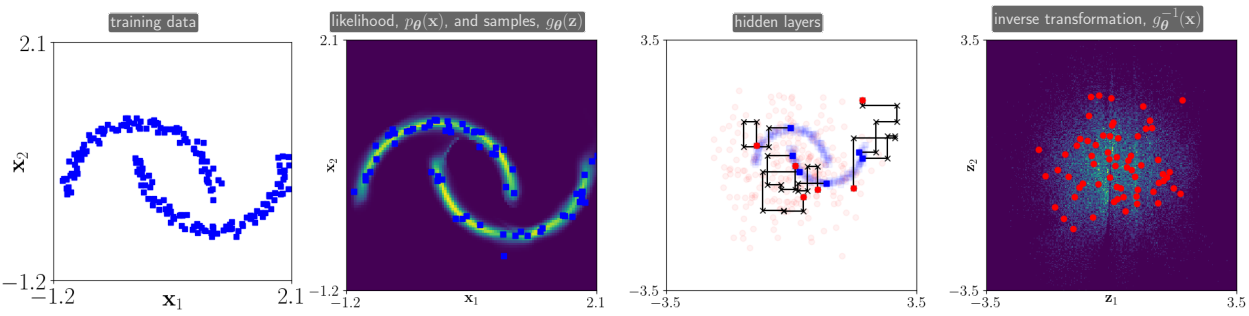


Figure 5: Normalizing flow results for the moon problem described in Example 1. Here, we use a real NVP approach with six hidden layers to transform samples from the standard normal distribution (red dots) to match the given data (blue squares). In the first subplot from the left, we show one batch of the training data. In the second subplot, we show the likelihood estimate superimposed by generated samples (blue square). Here, it is worth noting that, due to the flow model's smoothness, the two half-moons appear to be connected. In the third subplot, we show the generator's hidden layers for six randomly chosen latent variables as inputs (red dots). Due to the alternating fashion of the layers, the transformation for every layer is limited to one of the coordinates. In the fourth subplot, we show a two-dimensional histogram of the inverse transformation applied to 50,000 samples from the moon data set, superimposed by a few randomly chosen examples. As expected, the latent variables do approximately, but not perfectly, match a Gaussian distribution; see, for example, a narrow blue line passing approximately vertically through the origin that separates the parts associated with each cluster of the dataset.

ReLU nonlinearity with a slope parameter of 0.01. Overall, the flow network has 205,848 trainable weights, which we train using the stochastic approximation scheme ADAM [27]. We perform 20,000 steps of maximum likelihood training, each approximating the gradient of the objective function using a minibatch containing 256 points sampled from the true distribution $\mathcal{X}$.

We show the result of the training in Figure 5.

The real NVP approach has also been applied to higher dimensional examples [12]. Here, the partitions of the variables are more involved, and a certain depth of the network is required to ensure full coupling between the variables.

## 3.3 Continuous Normalizing Flows

While we can compute finite normalizing flows efficiently only for a specific choice of layers, we can obtain more flexibility in the framework of continuous normalizing flows (CNF). In a CNF [20], we define the generator as $g_{\boldsymbol{\theta}}(\mathbf{z}) = \mathbf{y}(T)$ where $T > 0$ is some terminal time and $\mathbf{y} : [0, T] \to \mathbb{R}^n$ satisfies the initial value problem

$$\partial_t \mathbf{y}(t) = v_{\boldsymbol{\theta}}(\mathbf{y}(t), t), \quad \text{where} \quad \mathbf{y}(0) = \mathbf{z}. \tag{14}$$

Here, $v_{\boldsymbol{\theta}} : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ is an arbitrary neural network parameterized by the weights $\boldsymbol{\theta} \in \mathbb{R}^{N_{\boldsymbol{\theta}}}$. For a sufficiently regular $v_{\boldsymbol{\theta}}$, the mapping $\mathbf{z} \mapsto \mathbf{y}(T)$ is invertible and in principle, one may define the inverse as $g_{\boldsymbol{\theta}}^{-1}(\mathbf{x}) = \mathbf{p}(0)$ where $\mathbf{p} : [0, T] \to \mathbb{R}^n$ satisfies the final value problem

$$-\partial_t \mathbf{p}(t) = v_{\boldsymbol{\theta}}(\mathbf{p}(t), t), \quad \text{where} \quad \mathbf{p}(T) = \mathbf{x}. \tag{15}$$

Here, we integrate backward in time as indicated by $-\partial_t$. While this process is straightforward in theory, it is important to note that the generator's stability and its inverse depends crucially on the design of $v_{\boldsymbol{\theta}}$, choice of weights, and the numerical integration used to solve (15). Simple integrators such as the commonly used forward Euler or even higher-order Runge-Kutta schemes without step size control can be prone to large errors when integrating backward; especially when the velocity in (14) changes rapidly along the curve $\mathbf{y}(\cdot)$.

There are several ways to avoid such issues. First, one may use symplectic integrators that can be reversed analytically up to machine precision. Second, as we will demonstrate in our experiment, we

can favor curves $\mathbf{y}(\cdot)$ that are simple (ideally, straight lines) by regularizing the velocity. This simplifies the integration and improves the inverse consistency. When using a non-conservative integrator we recommend also monitoring the inverse errors $\|g_{\boldsymbol{\theta}}^{-1}(g_{\boldsymbol{\theta}}(\mathbf{z})) - \mathbf{z}\|$ and $\|g_{\boldsymbol{\theta}}(g_{\boldsymbol{\theta}}^{-1}(\mathbf{x})) - \mathbf{x}\|$.

To compute the logarithm of the determinant of $g_{\boldsymbol{\theta}}$ we employ the Jacobi identity also used in [51, 20] and obtain

$$\log\det\nabla g_{\boldsymbol{\theta}}(\mathbf{x})^{-1} = \int_0^T -\mathrm{trace}\left(\nabla_{\mathbf{y}} v_{\boldsymbol{\theta}}(\mathbf{p}(t),t)\right) dt. \tag{16}$$

In practice, this computation can be combined with the numerical approximation of the characteristics (15).

**Relation to Optimal Transport** To shed more light into the CNF problem, we point out its similarities and differences to optimal transport [14, 49, 39]. To this end, we take a macroscopic view on the acting of the generator defined in (14) on the latent distribution $\mathcal{Z}$.

Let us denote the density function associated with $\mathcal{Z}$ by $\rho_0$, that is, $\rho_0$ is the density function of a univariate Gaussian. Then, the push forward of $\rho_0$ under the transformation that maps $\mathbf{z}$ to $\mathbf{y}(\tau)$ by integrating (14) until some $\tau \in [0,T]$ is given by $\rho(\cdot,\tau)$, the solution to the continuity equation

$$\partial_t \rho(\mathbf{x},t) + \nabla\cdot(\rho(\mathbf{x},t) v_{\boldsymbol{\theta}}(\mathbf{x},t)) = 0, \quad \rho(\mathbf{x},0) = \rho_0(\mathbf{x}). \tag{17}$$

Here, we see that the neural network $v_{\boldsymbol{\theta}}$ takes the role of an non-stationary velocity. We also note that (14) computes the characteristic curve originating in $\mathbf{z}$ forward in time and, similarly, (15) computes the same curve backward in time from point $\mathbf{x}$.

We can now formulate the CNF problem as a PDE-constrained optimization problem

$$\min_{\boldsymbol{\theta},\rho} \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[-\log\rho(\mathbf{x},T)\right] \quad \text{subject to} \quad (17). \tag{18}$$

Since the objective function solely depends on the density at the final time, the above problem does not attain a unique solution. To be precise, all velocity fields with the same initial and endpoints of the characteristics are assigned the same function value. Noting that (14) and (15) are trivial to solve when the velocity does not change along the characteristics, motivates us to add the $L_2$ transport cost and consider the regularized problem

$$\min_{v,\rho} \int \frac{1}{2}\|v_{\boldsymbol{\theta}}(\mathbf{x},t)\|^2 \rho(\mathbf{x},t) d\mathbf{x}dt + \alpha\mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[-\log\rho(\mathbf{x},T)\right] \quad \text{subject to} \quad (17), \tag{19}$$

where $\alpha > 0$ is a regularization parameter that balances between minimizing the transport costs and maximizing the log likelihood. We view this problem as a relaxed version of the dynamic optimal transport formulation [5] or more precisely as a mean field game [43]. The key difference to standard optimal transport settings is that the target density is unknown. From the optimal transport theory, it follows that (19) attains a unique solution for which the characteristics are straight lines. We note (19) can be reformulated into a convex optimization problem and can be solved efficiently using PDE constrained optimization techniques in dimensions $n \leq 3$ [22].

Several approaches that add transport costs to the CNF problem have been proposed recently [51, 50, 15, 32, 36]. While these approaches differ in some factors, including the definition of the objective function, network design, and numerical implementation, they provide ample numerical evidence to suggest that optimal transport techniques improve the training of the CNF. Since the methods are applied to machine learning benchmark datasets of tens or hundreds of dimensions, all numerical schemes rely on neural network parameterizations of the velocity and compute an approximate solution to (19) using stochastic approximation techniques. There is some numerical evidence that penalizing violations of the Hamilton-Jacobi-Bellman (HJB) equations, which are the necessary and sufficient first-order optimality conditions of (19), improves the practical performance [50, 36].

**Numerical Example: OT-Flow for Moons Dataset** We apply the OT-Flow approach introduced in [36] to the moons problem; see Example 1. This approach involves a Lagrangian PDE solver to solve the continuity equation (17) in a mesh-free manner, which renders the scheme scalable to high dimensions. Following the optimal transport theory, the approach defines the dynamics of the flow as the gradient of a potential, that is, $v_{\boldsymbol{\theta}}(\mathbf{x},t) = -\nabla\Phi_{\boldsymbol{\theta}}(\mathbf{x},t)$. Further, it has been shown empirically that adding

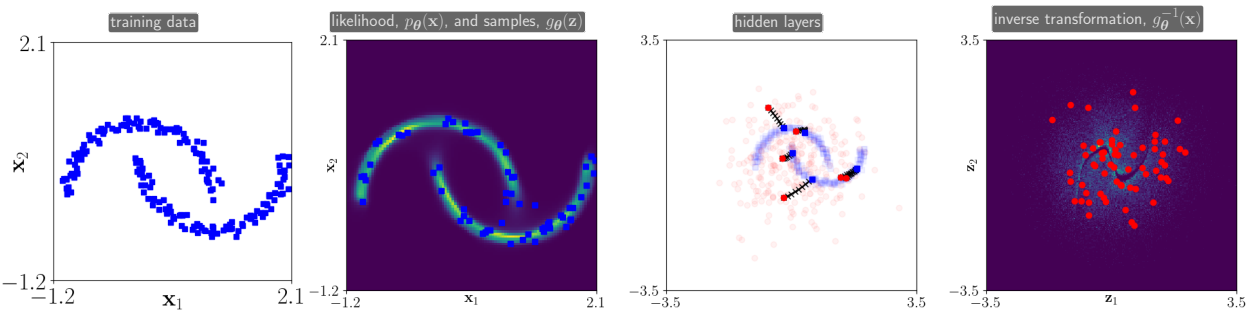


Figure 6: Continuous normalizing flow results for the moon problem described in Example 1. Here, we use the OT-Flow approach [36] to transform samples from the standard normal distribution (red dots) to match the given data (blue squares). In the first subplot from the left, we show one batch of the training data. In the second subplot, we show the likelihood estimate superimposed by generated samples (blue square). Despite the smoothness of the model, the two half-moons appear almost disconnected. In the third subplot, we show the hidden layers of the generator for six randomly chosen latent variables as inputs (red dots). Since the training is regularized by the transport costs, the characteristics are almost straight lines, which allows to invert the flow by integrating backwards in time. In the fourth subplot, we show a two-dimensional histogram of the inverse transformation applied to 50,000 samples from the moon data set, superimposed by a few randomly chosen examples. As expected, the latent variables do approximately, but not entirely, match a Gaussian distribution. In particular, note the narrow gap in the center of the domain that separates the parts associated with each cluster of the dataset.

a penalty function that enforces the Hamilton-Jacobi-Bellman (HJB) equations along the characteristic curves improves performance.

We parameterize $\Phi_{\boldsymbol{\theta}}$ as the sum of a quadratic form and a two-layer residual network whose second layer has 32 neurons; see [36] for details. This model has 1,229 trainable parameters, around two orders of magnitude fewer than the real NVP model used above. During the training, we compute the characteristics using a fourth-order Runge-Kutta scheme with equidistant time steps. As in the real NVP example, we train the network using 20,000 training steps of the ADAM scheme, each based on a minibatch containing 256 randomly sampled points.

We show the training results in Figure 6. Here, the trained flow provides meaningful samples from the dataset as well as realistic density estimates. Due to the penalization of transport costs, the characteristics are almost straight, which also helps reduce the inverse error nearly to machine precision.

# 4 Variational Autoencoders

In most practical situations, we cannot assume that the latent space dimension and that of the data space are equal. This prohibits a direct use of the flow models from the previous section since the generator is not invertible and the KL divergence may be unbounded or not well-defined [3]. Variational Autoencoders (VAE) [29, 42, 30] are a popular framework to overcome this limitation and, typically, use a latent space of much smaller dimension than the data space, that is, $q \ll n$. They also allow better control of the latent space dimension as we see next. Since the generator, $g_{\boldsymbol{\theta}}$, is not invertible, we cannot compute the negative log-likelihood loss using (8) directly.

Recall that we denote likelihood of a sample $\mathbf{x} \sim \mathcal{X}$ (also called its evidence) implied by the generator as $p_{\boldsymbol{\theta}}(\mathbf{x})$. Note that, using Bayes's rule, the likelihood can be re-written as

$$p_{\boldsymbol{\theta}}(\mathbf{x}) = \frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})} = \frac{p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})p_{\mathcal{Z}}(\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}, \quad \text{for,} \quad \mathbf{z} \sim \mathcal{Z}. \tag{20}$$

Recall the idea of maximum likelihood training from the previous section; that is, maximizing this likelihood with respect to $\boldsymbol{\theta}$. We note that directly maximizing the likelihood using the above expression

is infeasible: while computing conditional probability of a data sample given a sample from the latent space, $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$, is straightforward, the opposite direction is non-trivial. That is, the posterior distribution $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$, which quantifies the likelihood of a particular latent variable $\mathbf{z}$ to produce the given data sample $\mathbf{x}$, is generally intractable. This is particularly true when the generator is non-linear and non-invertible, which is the idea in deep generative modeling where $g_{\boldsymbol{\theta}}$ is a deep neural network.

In VAE, we use a variational inference approach to approximate the posterior $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ within a family of parameterized probability distributions that are tractable; that is, we can sample from the distribution and compute probabilities efficiently. In practice, the parameters of that distribution are given by the output of a second neural network. This allows us to define the approximate posterior

$$e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x}) \approx p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}). \tag{21}$$

Here, $\boldsymbol{\psi} \in \mathbb{R}^k$ are the weights of the neural network that provides the parameters of the approximate posterior. This network takes $\mathbf{x}$ as its input and yields the parameters of the approximate posterior, usually its mean, covariance and/or other parameters that determine a particular distribution. To enable efficient training, we must be able to evaluate the approximate posterior and draw samples efficiently. As is common, we use the same network weights, $\boldsymbol{\psi}$ for all $\mathbf{x}$, which is also called amortized inference.

We briefly remark that $e_{\boldsymbol{\psi}}$ acts similarly to an encoder in traditional autoencoders in that it maps from the data space $\mathcal{X}$ to the latent space $\mathcal{Z}$. However, a crucial difference to autoencoders is that this mapping is probabilistic; that is, rather than providing a single point in $\mathcal{Z}$, $e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})$ defines a probability distribution. One can view a point in this distribution, for example, the mean, as the result of an encoder. This construction is motivated by the non-invertibility of the generator caused by the difference between the data and latent space dimensions and its nonlinearity.

The importance of the posterior distribution, $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$, and its approximation also provides links to Bayesian inverse problems. Here, given a sample from the dataset, $\mathbf{x}$, the goal is to characterize the distribution of $\mathbf{z}$ in the latent space. Since VAE approaches seek to determine which latent vectors likely gave rise to the observation and use a deep network to approximate the posterior, they are also called deep latent variable models. In contrast to some applications of Bayesian inverse problems, there is a clear choice of the prior distribution in a VAE; namely, the prior distribution is $\mathcal{Z}$.

## 4.1 Evidence Lower Bound Training

Instead of maximizing the likelihood (20), we consider a tractable surrogate problem that we obtain by replacing the true posterior, $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ with the approximation $e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})$. For this approach to be meaningful, we have to accomplish two goals: maximize the approximate likelihood and reduce the approximation error in (21). As we illustrate now, the surrogate problem satisfies these two objectives since the approximate posterior yields a lower bound on the evidence $p_{\boldsymbol{\theta}}(\mathbf{x})$ and its maximization tightens the bound by reducing the approximation error in (21). We follow the argument in [30] to see that

$$\begin{aligned}
\log p_{\boldsymbol{\theta}}(\mathbf{x}) &= \mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x})\right] \\
&= \mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right] \\
&= \mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\cdot\frac{e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right] \\
&= \mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{p_{\boldsymbol{\theta}}(\mathbf{x},\mathbf{z})}{e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\right)\right] + \mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[\log\left(\frac{e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}{p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})}\right)\right]
\end{aligned}$$

Here, the second term is the KL divergence between the approximate posterior and the true posterior, $\mathrm{KL}(e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})||p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x}))$. Recall that the KL divergence is non-negative and zero only when equality holds in (21). Due to its non-negativity, dropping the KL divergence provides a lower bound on the evidence $p_{\boldsymbol{\theta}}(\mathbf{x})$. Therefore, the first term is also known as the *variational lower bound* or *evidence lower bound* (ELBO). To learn the weights $\boldsymbol{\psi}$ and $\boldsymbol{\theta}$ from samples, we minimize the negative of the ELBO and define

the loss

$$
\begin{aligned}
J_{\mathrm{ELBO}}(\boldsymbol{\psi}, \boldsymbol{\theta}) &= -\mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})\right] \\
&= \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[-\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z}) - \log p_{\mathcal{Z}}(\mathbf{z}) + \log e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})\right] \qquad (22) \\
&\approx \frac{1}{s}\sum_{i=1}^{s}\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x}^{(i)})}\left[-\log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}|\mathbf{z}) - \log p_{\mathcal{Z}}(\mathbf{z}) + \log e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x}^{(i)})\right], \qquad (23)
\end{aligned}
$$

with i.i.d. samples $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(s)}$ from $\mathcal{X}$. To avoid the intractability associated with $\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x}^{(i)})}$, we minimize $J_{\mathrm{ELBO}}$ using stochastic approximation schemes, where the expected value is approximated using a few (in practice only one) sample from the approximate posterior.

It is important to note that minimizing $J_{\mathrm{ELBO}}$ with respect to $\boldsymbol{\psi}$ improves the tightness of the bound as it simultaneously reduces the KL divergence between the approximate and true posterior. However, it is also worth noting that it is impossible to determine the tightness of the lower bound in practice due to the intractability of $p_{\boldsymbol{\theta}}(\mathbf{z}|\mathbf{x})$ and unknown $p_{\boldsymbol{\theta}}(\mathbf{x})$.

To gain further insight into the objective function in the VAE, we re-write (22) equivalently as

$$
\begin{aligned}
J_{\mathrm{ELBO}}(\boldsymbol{\psi}, \boldsymbol{\theta}) &= \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[-\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right] + \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[\log e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x}) - \log p_{\mathcal{Z}}(\mathbf{z})\right] \\
&= \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}\left[-\log p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})\right] + \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[\mathrm{KL}\left(e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})||p_{\mathcal{Z}}(\mathbf{z})\right)\right].
\end{aligned}
$$

Minimizing the first term reduces the approximation error in the data space that is introduced by restricting the dimension of the latent space and the approximation error introduced by the approximate posterior. Improving the approximate posterior, for example, reduces this term by providing samples $\mathbf{z}$ such that $g_{\boldsymbol{\theta}}(\mathbf{z})$ is more likely to be close to the given $\mathbf{x}$. Similarly, ensuring that generator's image contains $\mathbf{x}$ will help reduce this term provided a reasonably accurate approximation of the posterior. The second term can be seen as a regularizer that biases the approximate posteriors toward the distribution of the latent variable. In our examples, when $\mathcal{Z}$ is a univariate Gaussian, this term favors approximate posteriors whose samples (for a randomly chosen $\mathbf{x}$) are close to the origin.

The above discussion exposes a conflict between minimizing the reconstruction error and biasing the approximate posteriors toward the latent distribution. When minimizing solely the reconstruction error, which is similar to the training of autoencoders, samples from the (approximate) posterior will generally not be normally distributed in the latent space. Therefore, new data points generated by sampling $\mathbf{z} \sim \mathcal{Z}$ and applying such generator $g_{\boldsymbol{\theta}}(\mathcal{Z})$ are expected to be of low quality. Similarly, minimizing the second term, which in the extreme will make all approximate posteriors equal to the latent distribution, is expected to result in a substantial reconstruction error.

In the Bayesian framework used to derive VAEs here and in most parts of the literature, the only way to balance between minimizing the reconstruction error and regularity of the approximate posteriors is by choosing the likelihood function, $p_{\boldsymbol{\theta}}(\mathbf{x}|\mathbf{z})$. Consider, for example, the Gaussian likelihood function (3). Here, we can choose $\sigma$ to balance the reconstruction error and the regularity of the samples from the approximate posteriors. For the Bernoulli likelihood function (4), there is no obvious way to increase or decrease the importance of the reconstruction error in the VAE training in the Bayesian setting. To overcome this limitation, one can leave the Bayesian world and interpret $J_{\mathrm{ELBO}}$ only as a regularized loss function. In this interpretation, one can use different reconstruction losses (including loss functions not related to probabilities) and various penalty terms that measure the discrepancy between the approximate posteriors (or their samples) and the latent distribution. While this venue provides exciting opportunities to improve upon standard VAEs, it is not clear a-priori that the regularized loss function will be a lower bound to the evidence.

## 4.2 Example: Gaussian Posterior

In the MNIST problem, we chose to approximate the posterior distribution with a Gaussian for computational convenience, that is,

$$
e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x}) = \mathcal{N}\left(\boldsymbol{\mu}_{\boldsymbol{\psi}}(\mathbf{x}), \exp(\boldsymbol{\Sigma}_{\boldsymbol{\psi}}(\mathbf{x}))\right). \qquad (24)
$$

The subscripts indicate that the value of the mean $\boldsymbol{\mu}$ and the logarithm of the covariance matrix $\boldsymbol{\Sigma}$ depend on the weights $\boldsymbol{\psi}$ and the input vector $\mathbf{x}$. It is common to model both using the same neural network that differs only in its last layer. Although one would expect a multi-modal posterior distribution given the nonlinearity of the generator, this simple model has been shown to be effective in some cases.

To enable learning the weights $\boldsymbol{\psi}$ using derivative-based minimization, a difficulty that arises is the differentiation of a sample $\mathbf{z} \sim e_\psi(\mathbf{z}|\mathbf{x})$ with respect to the weights $\boldsymbol{\psi}$. This obstacle can be overcome using the so-called reparametrization trick, where we write

$$\mathbf{z}(\boldsymbol{\epsilon}) = \boldsymbol{\mu}_{\boldsymbol{\psi}} + \exp(\boldsymbol{\Sigma}_{\boldsymbol{\psi}}(\mathbf{x}))\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}). \tag{25}$$

This allows replacing the expectation $\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}$ with the expectation $\mathbb{E}_{\boldsymbol{\epsilon}\sim p(\boldsymbol{\epsilon})}$ and enables the use of Monte Carlo estimation during the training. We compute

$$\nabla_{\boldsymbol{\psi}} J_{\mathrm{ELBO}}(\boldsymbol{\psi}, \boldsymbol{\theta}) = -\nabla_{\boldsymbol{\psi}} \mathbb{E}_{\mathbf{x}\sim\mathcal{X}} \mathbb{E}_{\boldsymbol{\epsilon}\sim p(\boldsymbol{\epsilon})} \left[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}(\boldsymbol{\epsilon})) - \log e_{\boldsymbol{\psi}}(\mathbf{z}(\boldsymbol{\epsilon})|\mathbf{x})\right]. \tag{26}$$

This re-parameterization provides an unbiased estimate of the gradient when the latent variable is continuous and the encoder and decoder are differentiable; see [30, Sec. 2.4] for details.

**Numerical Example: VAE for MNIST Example** We train the generator for the MNIST problem in Example 2 using the VAE approach. Recall that we defined the latent space to be two-dimensional. Since the image intensities are in $[0, 1]$, we measure the reconstruction quality using the Bernoulli likelihood (4); see also [29, Appendix C.1]. We use the same architecture of the neural network used to compute the mean and covariance of the approximate posterior as in the excellent VAE tutorial [45], but note that our generator is different.

For a given MNIST image $\mathbf{x}$ we use two convolution layers for feature extraction

$$\begin{aligned} \mathbf{h}^{(1)} &= \sigma_{\mathrm{ReLU}}\left(\mathbf{C}^{(1)}_{\mathrm{VAE}}\mathbf{x} + \mathbf{c}^{(1)}_{\mathrm{VAE}}\right) \\ \mathbf{h}^{(2)} &= \sigma_{\mathrm{ReLU}}\left(\mathbf{C}^{(2)}_{\mathrm{VAE}}\mathbf{h}^{(1)} + \mathbf{c}^{(2)}_{\mathrm{VAE}}\right). \end{aligned} \tag{27}$$

Here, $\mathbf{C}^{(1)}_{\mathrm{VAE}}$ and $\mathbf{C}^{(2)}_{\mathrm{VAE}}$ are convolution operators with $4 \times 4$ stencils and strides of two, that is, they reduce the number of pixels by a factor of two in each axis. The first layer has 32 hidden channels and the second layer has 64 hidden channels. The bias vectors $\mathbf{c}^{(1)}_{\mathrm{VAE}}$ and $\mathbf{c}^{(2)}_{\mathrm{VAE}}$ apply constant shifts to each channel. Given the feature $\mathbf{h}^{(2)}$, we compute the mean and the diagonal of the covariance of the approximate posterior, $e_\psi(\mathbf{z}|\mathbf{x})$, using

$$\boldsymbol{\mu}_{\boldsymbol{\psi}}(\mathbf{x}) = \mathbf{D}^{(1)}_{\mathrm{VAE}}\mathbf{h}^{(2)} + \mathbf{d}^{(1)}_{\mathrm{VAE}} \quad \text{and} \quad \mathrm{diag}(\boldsymbol{\Sigma}_{\boldsymbol{\psi}}(\mathbf{x})) = \mathbf{D}^{(2)}_{\mathrm{VAE}}\mathbf{h}^{(2)} + \mathbf{d}^{(2)}_{\mathrm{VAE}}, \tag{28}$$

where $\mathrm{diag}(\boldsymbol{\Sigma}_{\boldsymbol{\psi}}(\mathbf{x}))$ denotes the diagonal entries of the matrix. The vector $\boldsymbol{\psi}$ collects all the trainable parameters in $\mathbf{C}^{(1)}_{\mathrm{VAE}}, \mathbf{C}^{(2)}_{\mathrm{VAE}}, \mathbf{c}^{(1)}_{\mathrm{VAE}}, \mathbf{c}^{(2)}_{\mathrm{VAE}}, \mathbf{D}^{(1)}_{\mathrm{VAE}}, \mathbf{D}^{(2)}_{\mathrm{VAE}}, \mathbf{d}^{(1)}_{\mathrm{VAE}}$, and $\mathbf{d}^{(2)}_{\mathrm{VAE}}$. The number of trainable weights in this network is 45,924.

We initialize the network weights using the default option in pytorch and then train the weights using the ADAM optimizer with a fixed learning rate of $10^{-3}$ for 50 epochs[2] with minibatches of size $s = 64$. We approximate the integrals for $\mathbb{E}_{\mathbf{z}\sim e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})}$ using Monte Carlo quadrature of a single sample. To regularize the weights, we use weight decay with parameter $10^{-5}$.

In Figure 7, we visualize the approximation error for four randomly selected images. While the approximate posterior is relatively close to a mode of the true posterior in both cases, the reconstruction quality in the top row is substantial, even showing an incorrect digit. These plots also suggest that the lower bound on the log-likelihood given by (22) is not very tight. Further improving the tightness of this bound may be possible with non-Gaussian models for $e_\psi$. One approach that increases the expressiveness of the approximate encoder using a continuous normalizing flow in the latent space is presented in [20]. As can be seen from Figure 7, the true posteriors vary drastically for each example. Hence, the weights of the flow typically depend on $\mathbf{x}$, for instance, introducing a third neural network.

We investigate the distribution of samples drawn from the approximate posteriors $e_\psi(\mathbf{z}|\mathbf{x}^{(j)})$ for all $\mathbf{x}^{(j)}$ in the test set in the left two subplots of Figure 8. The left subplot shows the mean of the approximate posteriors colored by the class label of the underlying image. Although we did not use the information about the digit shown in the image, this plot shows that the embedding performs a rough clustering for some classes. The second plot from the left is a two-dimensional histogram of samples from the approximate posteriors. For each test image, we sample ten points from the approximate posterior. This plot shows that the approximate posteriors collectively do not overlap with a Gaussian and that there are regions in the center of the domain with relatively few samples.

[2]An epoch is one pass through the entire training data set

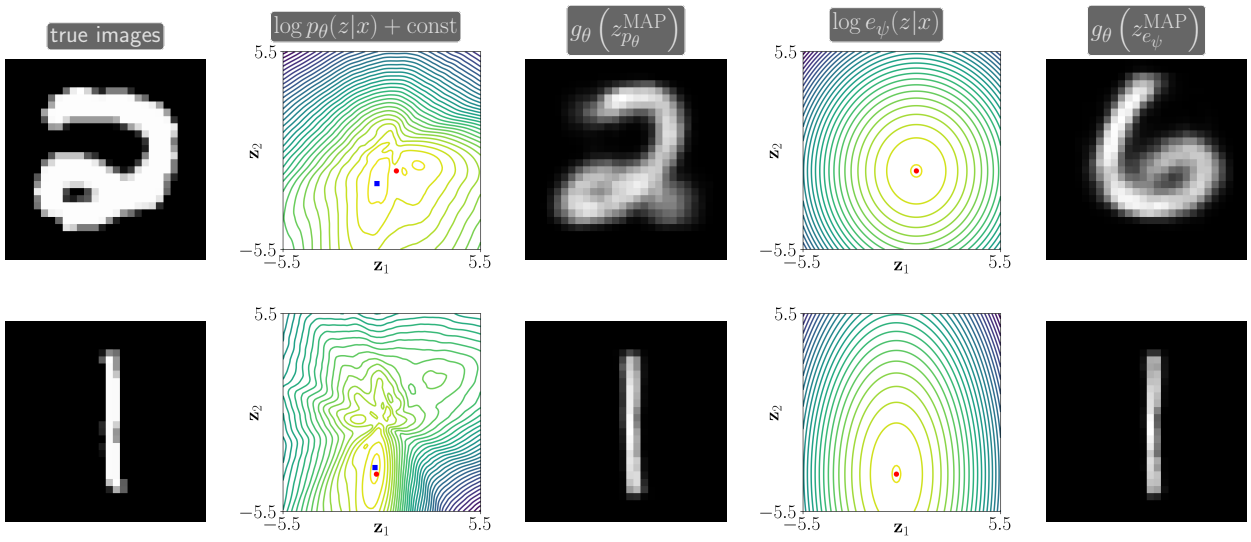


Figure 7: Illustrating the error caused by the approximated posterior (21) in the variational autoencoder for two example images from the MNIST dataset (row-wise). For the images in the left column, we compute the log posterior (up to a constant shift) on a rectangular grid and depict its maximum a posteriori (MAP) estimate with a blue square (second column). The third column shows the reconstruction associated with the MAP estimate, which looks comparable to the true image in both cases. For these examples, the approximate posteriors and their MAP points (red dots in the second and fourth column) are slightly different from the true posterior. The impact of this error is different in both cases. In the example shown in the first row, we observe a substantial reconstruction error (right column) leading to an incorrect digit. The situation is slightly better for the example in the bottom row, where the reconstruction error is minimal. However, we note that the true posteriors in both cases are far from being Gaussian. This motivates the use of more sophisticated approximate posteriors.

While the goal in VAE is to train the generator $g_{\boldsymbol{\theta}}$ such that it maps samples from the latent distribution to the data distribution, we note that we sample the latent variable from the approximate posteriors, $e_{\boldsymbol{\psi}}(\mathbf{z}|\mathbf{x})$ during training. For these samples, we train the generator to minimize the reconstruction error, which should provide realistic images. This raises the concern that the quality of images generated from points $\tilde{\mathbf{z}}$ at which $p_{\mathcal{Z}}(\tilde{\mathbf{z}}) \gg e_{\boldsymbol{\psi}}(\tilde{\mathbf{z}}|\mathbf{x})$ will be poor. To investigate this further, we use the histogram plot shown in the second subplot of Figure 8 as an approximate density and compare it to the prior density. Of the 2000 points with the largest difference, we randomly choose 64 (indicated as red dots) and visualize the generated images in the rightmost subplot. We order the images by the first component of the latent variable, $\mathbf{z}_1$, from the top left to the bottom right. While we do not expect all images to look realistic since the generator rarely visited these points during training, overall, the quality of the samples is comparable to completely random samples. In this batch, the sample quality does not appear to correlate with $\mathbf{z}_1$, which is surprising given the sparsity of samples in the fourth quadrant. Even though the generator seems to be effective in regions not visited during training for this example, we recall that MNIST is known to be a relatively simple dataset. Hence, we do not entirely reject the concern about the difference between the distributions used during training and evaluation.

In the left column of Figure 9, we show a few random samples from the dataset (top) and from the generator trained using the VAE approach (bottom). The most striking difference between the true and the generated images is the apparent blur in the latter ones. Despite the blur, one can recognize a hand-written digit in most of the generated images.

In the top row of Figure 10, we use the trained generator to interpolate images along the line segment from $-\mathbf{e} = (-1, -1)^\top$ to $\mathbf{e} = (1, 1)^\top$ in the latent space in equidistant steps. While the images appear slightly blurred, we can recognize most of them as hand-written digits. It is noteworthy that the generator

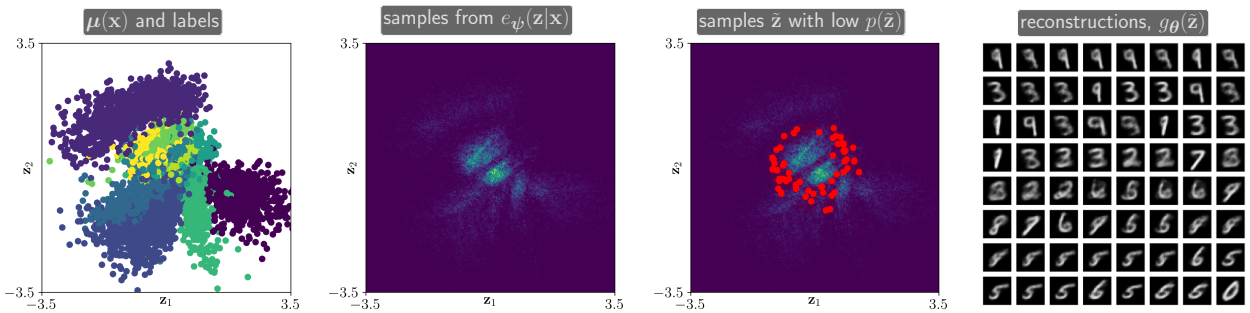


Figure 8: Visualizing the structure of the VAE latent space for the MNIST example. The first subplot from the left shows the means of the approximate posteriors for the 10,000 test images color-coded by their class label. Some classes are clustered even though we did not use the labels during training. The second subplot shows a two-dimensional histogram of samples from the approximate posteriors (10 samples each), which shows striking differences to samples from a standard normal distribution. In the third subplot, we superimpose the histogram with red dots that mark 64 randomly chosen points for which the prior probability is large but where few samples from the approximate posteriors are located. The fourth subplot shows the generated images from those points ordered increasingly by their $\tilde{\mathbf{z}}_1$ component (starting in the top left, ending in the bottom right).

produces images showing different digits, which are far apart in the data space.

# 5 Generative Adversarial Networks

In generative adversarial networks (GAN), we train the weights of $\boldsymbol{\theta}$ by minimizing a loss function that measures the distance between $g_{\boldsymbol{\theta}}(\mathcal{Z})$ and $\mathcal{X}$; that is, GANs compare the distributions in the data space, unlike CNFs and VAEs. Recall that $\mathcal{X}$ is represented by the training data and samples from $g_{\boldsymbol{\theta}}(\mathcal{Z})$ obtained by transforming samples of the latent distribution. GANs are considered likelihood-free models since they neither use the samples' likelihood (as in the normalizing flows discussed in Section 3) nor using a lower bound of the likelihood (as in the variational autoencoder presented in Section 4). Another difference to the previous approaches is that GANs do not attempt to infer the latent variables that underlie the samples. Many promising results [8, 26] have contributed to GANs' increasing popularity, and several excellent works that go beyond our short presentation are [19, 17, 2].

A key challenge in GAN training is to define a loss function that effectively measures the distance between $g_{\boldsymbol{\theta}}(\mathcal{Z})$ and $\mathcal{X}$ from samples that have no known correspondence. The objective function must also allow effective approximation using minibatches of small or modest size to enable efficient optimization. In the following, we discuss two standard options to define the distance function in a GAN. Both involve a second, scalar-valued neural network, often called the discriminator, that introduces another set of weights to the training problem. In both cases, training the weights of the generator and discriminator results in a saddle point problem that, not surprisingly, is challenging to solve. The saddle point problem can be interpreted as a two-player non-cooperative game between the generator and the discriminator network.

## 5.1 Discriminators based on Binary Classification

GANs were popularized by the seminal work of Goodfellow et al. [19] that casts GAN training as a two-sample test problem. Here, the discriminator, $d_{\boldsymbol{\phi}} : \mathbb{R}^n \to [0, 1]$, is trained to predict the probability that a given example was part of the training dataset. This leads to a binary classification problem for the discriminator; that is, we seek to choose $\boldsymbol{\phi}$ such that $d_{\boldsymbol{\phi}}(\mathbf{x}) \approx 1$ when $\mathbf{x} \sim \mathcal{X}$ and $d_{\boldsymbol{\phi}}(\mathbf{x}) \approx 0$ when $\mathbf{x} \sim g_{\boldsymbol{\theta}}(\mathcal{Z})$. Here, it is important to recall that we only sample from $g_{\boldsymbol{\theta}}(\mathcal{Z})$ but do not attempt to estimate the likelihood $p_{\boldsymbol{\theta}}(\mathbf{x})$. Clearly, the training of the discriminator is coupled with the training of the generator whose goal it is to provide samples that are indistinguishable from the true dataset.

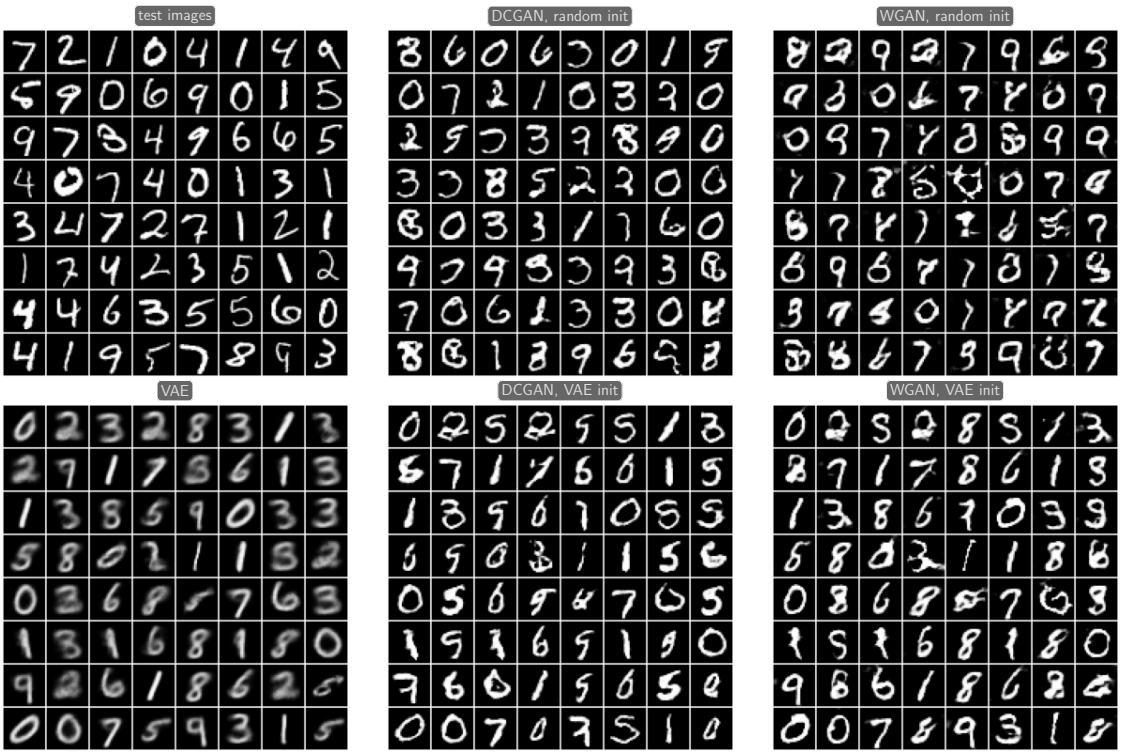


Figure 9: Comparison of true MNIST images (top left) to randomly drawn samples from the trained VAE, DCGAN, and WGAN (first, second, and third column, respectively). While most samples from the generator trained using the VAE framework clearly show hand-written digits, the images tend to be blurry. While the best images generated using the GAN approaches are indistinguishable from the true MNIST images, there are more samples in which no hand-written digit is shown. The human eye can easily detect those as fakes.

Due to the relation to binary classification, it is common to train the GAN's generator and discriminator using the cross-entropy loss function

$$J_{\mathrm{GAN}}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \mathbb{E}_{\mathbf{x}\sim\mathcal{X}}\left[\log(d_{\boldsymbol{\phi}}(\mathbf{x}))\right] + \mathbb{E}_{\mathbf{z}\sim\mathcal{Z}}\left[\log\left(1 - d_{\boldsymbol{\phi}}(g_{\boldsymbol{\theta}}(\mathbf{z}))\right)\right]. \tag{29}$$

The discriminator seeks to maximize this function (indicating low classification errors) while the generator seeks to minimize this function (corresponding to a confused discriminator). In other words, training the weights of the generator and discriminator using the loss function is equivalent to finding a Nash equilibrium $(\boldsymbol{\theta}^*, \boldsymbol{\phi}^*)$ such that

$$\boldsymbol{\phi}^* \in \arg\max_{\boldsymbol{\phi}} J_{\mathrm{GAN}}(\boldsymbol{\theta}^*, \boldsymbol{\phi}) \quad \text{and} \quad \boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta}} J_{\mathrm{GAN}}(\boldsymbol{\theta}, \boldsymbol{\phi}^*). \tag{30}$$

To gain some appreciation for the difficulty of this problem, consider, for example, that $\boldsymbol{\theta}^*$ are the weights of the optimal generator, that is, $g_{\boldsymbol{\theta}^*}(\mathcal{Z}) = \mathcal{X}$. In this case, the discriminator will be maximally confused, and $d_{\boldsymbol{\phi}^*}$ would predict $\frac{1}{2}$ for all samples. However, note that saddle points (as opposed to minimizers) are unstable and very difficult to approximate numerically. Hence, for a slightly suboptimal generator, the discriminator can significantly increase the objective by learning to distinguish between the training data and generated samples and vice versa. Since the expressiveness of both the generator and discriminator are limited by their parameterization, the effectiveness of a GAN is almost impossible to predict a priori; for more detailed theory on this and other issues, we refer to [2].

In practice, it is common to solve problem (30) approximately in an alternating fashion. The choice of the iterative method crucially impacts the performance. As a simple example, consider a stochastic

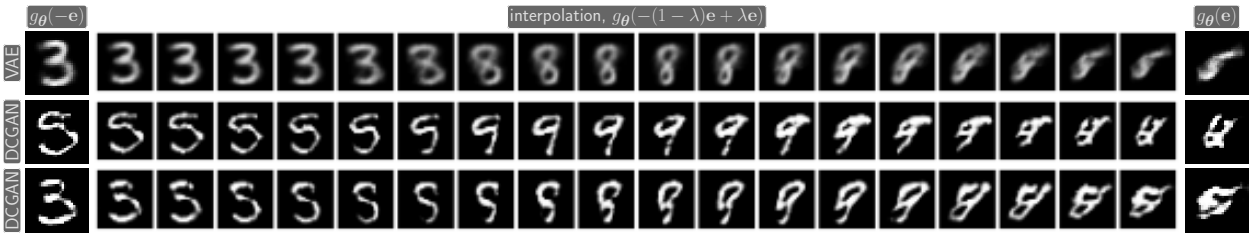


Figure 10: Interpolation across the latent space for the MNIST example. For the VAE, DCGAN, WGAN, we interpolate between $-\mathbf{e} = (-1,-1)^\top$ and $\mathbf{e} = (1,1)^\top$ in 20 equidistant steps. While the images from the VAE example (top row) are blurred, a digit can be recognized in most of them. While the GAN samples appear sharper, not all of them resemble MNIST images, that is, some do not resemble any digit.

gradient scheme whose $k$th step reads

$$\boldsymbol{\phi}^{(k+1)} = \boldsymbol{\phi}^{(k)} + \gamma_{\boldsymbol{\phi}}^{(k)} \frac{1}{s} \sum_{i=1}^{s} \left[ \nabla_{\boldsymbol{\phi}} \log(d_{\boldsymbol{\phi}^{(k)}} \left( \mathbf{x}^{(i)} \right)) + \nabla_{\boldsymbol{\phi}} \log \left( 1 - d_{\boldsymbol{\phi}^{(k)}} (g_{\boldsymbol{\theta}^{(k)}} (\mathbf{z}^{(i)})) \right) \right], \tag{31}$$

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \gamma_{\boldsymbol{\theta}}^{(k)} \frac{1}{s} \sum_{i=1}^{s} \nabla_{\boldsymbol{\theta}} \log \left( 1 - d_{\boldsymbol{\phi}^{(k+1)}} (g_{\boldsymbol{\theta}^{(k)}} (\mathbf{z}^{(i)})) \right), \tag{32}$$

where the samples $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(s)}$ and $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \ldots, \mathbf{z}^{(s)}$ are i.i.d. and re-sampled in every step and we have dropped the term associated with the true examples in the second line as it is independent of $\boldsymbol{\theta}$. Here, $\gamma_{\boldsymbol{\phi}}^{(k)}$ and $\gamma_{\boldsymbol{\theta}}^{(k)}$ are learning rates that are typically chosen a priori by the user. Empirically, SGD variants such as ADAM [27], and RMSProp [48] are typically more efficient and lead to better solutions than the plain stochastic gradient scheme shown above.

The critical hurdle during training is balancing between the two subproblems in (30). Consider, for example, the beginning of training, when it is relatively easy to distinguish the actual and generated samples. On the one hand, training the discriminator to optimality would make it impossible for the generator to improve, since the gradient $\nabla_{\boldsymbol{\theta}} J_{\mathrm{GAN}}(\boldsymbol{\theta}, \boldsymbol{\phi}^*)$ would be close to zero. On the other hand, not training the discriminator well enough would make it challenging to update the weights of the generator.

Another common problem that also presents theoretical challenges to the GAN formulation is known as mode collapse. To gain some intuition, consider the extreme case when the generator maps the entire distribution $\mathcal{Z}$ to a single data point, say $\mathbf{x}^{(1)} \sim \mathcal{X}$, that is,

$$g_{\boldsymbol{\theta}}(\mathbf{z}) = \mathbf{x}^{(1)} \quad \text{for almost all} \quad \mathbf{z} \sim \mathcal{Z}.$$

In this case, the optimal discriminator would yield $d_{\boldsymbol{\phi}^*}(\mathbf{x}^{(1)}) = \frac{1}{2}$ and $d_{\boldsymbol{\phi}^*}(\mathbf{x}^{(j)}) = 1$ for all $j > 1$ and the training would terminate.

It is important to note that we can easily detect the above example of mode collapse by inspecting a few samples from the generator, which will all be identical. A more difficult case would be when the generator mapped almost no point from $\mathcal{Z}$ close to $\mathbf{x}^{(1)}$. If the data set contains a few thousand data points or more, such failure is almost impossible to detect by analyzing a finite number of generated samples. Several heuristics have been proposed to reduce the risk of mode collapse; for example, one can add distance terms that compare the statistics of the minibatches or apply one-sided label smoothing [44].

**Numerical Experiment: DCGAN for MNIST** We continue our MNIST Example 2 and seek to train the generator along with a discriminator whose architecture is similar to the one used in Deep Convolutional GAN (DCGAN) [44]. To be specific, we define the discriminator using two convolution layers and one fully connected layer: that is, given the input feature $\mathbf{x} \in \mathbb{R}^n$, we predict the probability

that $\mathbf{x}$ is sampled from the true dataset using

$$\begin{aligned}\mathbf{v}^{(1)} &= \sigma_{\ell\mathrm{ReLU}}\left(\mathcal{N}\left(\mathbf{C}^{(1)}_{\mathrm{GAN}}\mathbf{x} + \mathbf{c}^{(1)}_{\mathrm{GAN}}\right)\right)\\ \mathbf{v}^{(2)} &= \sigma_{\ell\mathrm{ReLU}}\left(\mathcal{N}\left(\mathbf{C}^{(2)}_{\mathrm{GAN}}\mathbf{v}^{(1)} + \mathbf{c}^{(2)}_{\mathrm{GAN}}\right)\right)\\ d_{\boldsymbol{\phi}}(\mathbf{x}) &= \sigma_{\mathrm{sigm}}\left(\left(\mathbf{d}_{\mathrm{GAN}}\right)^{\top}\mathbf{v}^{(2)} + \delta_{\mathrm{GAN}}\right)\end{aligned} \tag{33}$$

Here, $\mathbf{C}^{(1)}_{\mathrm{GAN}}$ and $\mathbf{C}^{(2)}_{\mathrm{GAN}}$ are convolution operators, $\mathbf{d}_{\mathrm{GAN}}$ is a vector, $\mathbf{c}^{(1)}_{\mathrm{GAN}}, \mathbf{c}^{(2)}_{\mathrm{GAN}}, \delta_{\mathrm{GAN}}$ are bias terms, $\mathcal{N}$ is a batch normalization layer, and $\sigma_{\ell\mathrm{ReLU}}$ is the leaky ReLU activation

$$\sigma_{\ell\mathrm{ReLU}}(x) = \begin{cases} x & x \geq 0 \\ 0.2 & \text{else} \end{cases}.$$

To abbreviate the notation, we collect the trainable parameters in $\mathbf{C}^{(1)}_{\mathrm{GAN}}, \mathbf{C}^{(3)}_{\mathrm{GAN}}, \mathbf{d}_{\mathrm{GAN}}, \mathbf{c}^{(1)}_{\mathrm{GAN}}, \mathbf{c}^{(2)}_{\mathrm{GAN}}, \delta_{\mathrm{GAN}}$ in the vector $\boldsymbol{\phi}$.

The first two layers contain the convolution operators $\mathbf{C}^{(1)}_{\mathrm{GAN}}$ and $\mathbf{C}^{(2)}_{\mathrm{GAN}}$, whose structure is identical to the operators $\mathbf{C}^{(1)}_{\mathrm{VAE}}$ and $\mathbf{C}^{(2)}_{\mathrm{VAE}}$ used in the VAE example. In addition to the different convolution stencils, the main difference here is the behavior of the activation function for negative entries in the feature vector. As in the VAE example, the output of the second layer is a vector of length $7 \cdot 7 \cdot 64$, which we multiply with $\mathbf{d}_{\mathrm{GAN}} \in \mathbb{R}^{64 \cdot 49}$ and shift by the scalar $\delta_{\mathrm{GAN}} \in \mathbb{R}$ before using the sigmoid function to obtain the final value.

In training, we perform the steps in (31) with gradients approximated using minibatches of size 64 and using the ADAM scheme. We use fixed learning rates of 0.0002 and, as proposed in [44], a momentum of 0.5. We observed that the training performance is highly dependent on these parameter choices and that, for instance, changes in the batch size can quickly lead to complete failure of the training. We perform a fixed number of 50,000 training steps.

We compare two ways of initializing the weights. First, we use the default random initialization implemented in pytorch for all the generator and discriminator weights. Second, we initialize the discriminator randomly as above but use the optimal weights from the VAE example in the generator.

We show random samples and samples obtained by interpolating across the latent space in the second column of Figure 9 and the middle row of Figure 10, respectively. The similarity of the samples to true MNIST images varies considerably. The best images are almost indistinguishable from the actual distribution, but many images do not appear to contain any of the digits.

As the GAN training seeks to find a saddle point of $J_{\mathrm{GAN}}$ in (29), monitoring its value during training does not provide useful insight into the quality of the generator. Often the best option is to inspect a few generated samples at some intermediate steps visually. As described above, this can be misleading, for example, due to mode collapse. To obtain some insight into the convergence of the method, we estimate the distance between $\mathcal{X}$ and $g_{\boldsymbol{\theta}}(\mathcal{Z})$ using the multivariate $\varepsilon$ test for equal distributions suggested in [47]. As can be seen in Figure 11, this distance is reduced during the GAN training both from the random initialization (blue dashed line) and when initializing the generator with the weights from the VAE training (solid blue line). The latter considerably reduces the number of training steps needed and obtains a better score overall.

## 5.2 Discriminators based on Transport Costs

The idea of Wasserstein GANs [3] is to use an approximation of the earth mover distance (EMD) to measure the distance between $g_{\boldsymbol{\theta}}(\mathcal{Z})$ and $\mathcal{X}$. The EMD distance, also known as Wasserstein-1 distance, can also be seen as the cost of the optimal transport plan that moves $g_{\boldsymbol{\theta}}(\mathcal{Z})$ to $\mathcal{X}$. The Wasserstein-1 distance can be written as

$$W_1(g_{\boldsymbol{\theta}}(\mathcal{Z}), \mathcal{X}) = \inf_{\gamma \in \Pi} \mathbb{E}_{(\widehat{\mathbf{x}}, \mathbf{x}) \sim \gamma}\left[\|\widehat{\mathbf{x}} - \mathbf{x}\|\right] \tag{34}$$

where $\Pi$ denotes the set of all distributions $\gamma(\widehat{\mathbf{x}}, \mathbf{x})$ whose marginals are $\mathcal{X}$ and $g_{\boldsymbol{\theta}}(\mathcal{Z})$, respectively. The value of $\gamma(\widehat{\mathbf{x}}, \mathbf{x})$ indicates how much mass is moved between the two locations, and the distance is measured using the Euclidean norm.

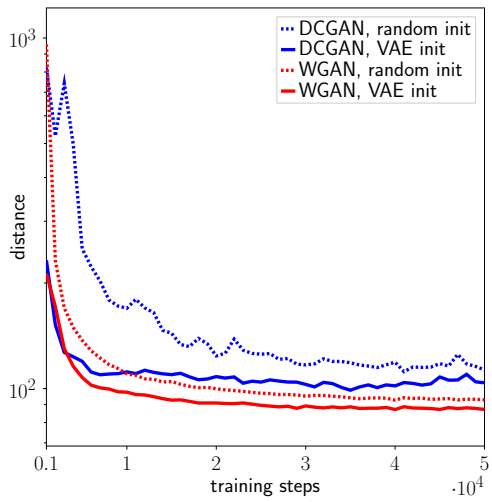


Figure 11: Estimating the distance between $\mathcal{X}$ and $g_{\boldsymbol{\theta}}(\mathcal{Z})$ for the MNIST example using the multivariate $\varepsilon$ test for equal distributions [47]. A lower value suggests an improved performance of the generator. The DCGAN (blue) and WGAN (red) approaches reduce this distance measure effectively when initialized randomly (dashed) and when starting from the VAE solution (solid). According to this measure, starting from the VAE solution improves the results overall and the WGAN slightly outperforms the DCGAN.

Instead of the formulation (34), most practical implementations of GANs use the equivalent formulation

$$W_1(g_{\boldsymbol{\theta}}(\mathcal{Z}), \mathcal{X}) = \max_{f \in \mathrm{Lip}(f) \leq 1} \mathbb{E}_{\mathbf{z} \sim \mathcal{Z}} \left[ f\left(g_{\boldsymbol{\theta}}(\mathbf{z})\right) \right] - \mathbb{E}_{\mathbf{x} \sim \mathcal{X}} \left[ f(\mathbf{x}) \right], \tag{35}$$

which is also known as the Kantorovich and Rubinstein norm; see [39] for more details. Here, the maximum is taken over all functions $f : \mathbb{R}^n \to \mathbb{R}$ that are Lipschitz-1 continuous. Computing such $f$ is far from trivial, especially in high dimensions. In the context of GANs it has become common to approximate the function $f$ with a neural network $f_{\boldsymbol{\phi}}$. We note, however, that ensuring the Lipschitz continuity of the neural network approximation is difficult.

Wasserstein GANs have several appealing theoretical advantages over the discriminator-based GANs, including the ones found in [3]. For example, the loss function is continuous as long as $g_{\boldsymbol{\theta}}$ is continuous and differentiable almost everywhere when $g_{\boldsymbol{\theta}}$ is locally Lipschitz continuous. However, it is unclear which of these advantages can be realized or is even desirable in practical applications. In fact there are examples in which training $f_{\boldsymbol{\phi}}$ and thus approximating $W_1(g_{\boldsymbol{\theta}}(\mathcal{Z}), \mathcal{X})$ more accurately can reduce the performance of the generator [46].

**Numerical Experiment: WGAN for MNIST** We continue the MNIST example and train the generator described in Example 2 using a WGAN approach. The architecture for the potential $f_{\boldsymbol{\phi}}$ is equal to the one used as the discriminator in (33) except that the sigmoid in the last layer is removed. Following [3] we impose bound constraints on $\boldsymbol{\phi}$ to regularize $f_{\boldsymbol{\phi}}$. This will, in general, not be sufficient to ensure the Lipschitz continuity. More promising (but also more involved) ways to incorporate this constraint is using gradient penalty methods [21] or spectral normalization [34].

We train the network using 50 iterations of RMSprop. As before, we perform two experiments. One uses the default random initialization in pytorch and one starts from the solution obtained by the VAE.

We show random samples and interpolated images, respectively, in the third column of Figure 9 and the bottom row of Figure 10 are qualitatively similar to the DCGAN images. The sharpness of the images resembles that of the true images, and many samples are very realistic. However, many images do not contain a hand-written digit and are thus easy for the human eye to be recognized as fakes.

Both for the random and the VAE initialization, the WGAN approach outperforms the DCGAN in terms of the multivariate $\varepsilon$ test [47]; see Figure 11. Again, initializing the generator with the VAE

solution leads to a quicker and overall larger reduction of this metric. It has to be noted that this test, like any other statistical test, may not agree with the human assessment of the samples; compare middle and right column in Figure 9. The lack of a useful metric is one of the main difficulties in training GANs. In contrast to normalizing flows and variational autoencoders, no information about the likelihood or the latent space is available.

# 6 Discussion of the three main approaches

This paper provided an introduction to deep generative modeling and the three currently dominating classes of training approaches. The goal of DGM training is to learn to generate examples that are "similar" to those obtained from an intractable distribution. This is done by approximating a complicated and generally high-dimensional probability from samples. The idea is to transform a known and simple distribution (for example, a univariate Gaussian) using a deep neural network that acts as a generator. A key challenge in training is the lack of correspondence between points in the latent space and the data space. This difficulty results in choosing an effective design and training of the generator. We focused most of our attention on different options to derive objective functions that help train the generator. We see this as the critical difference between today's most commonly used DGM approaches. Our goal is to establish a mathematical and practical understanding of these approaches and motivate the reader to explore this topic further.

**Finite and Continuous Normalizing Flows** Continuous normalizing flows (CNF) have several advantages over the variational autoencoder (VAE) and generative adversarial network (GAN) approaches. First, by assuming a diffeomorphic generator, CNFs can directly compute and optimize the likelihood of the data samples, which alleviates the difficulty of quantifying the similarity of $\mathcal{X}$ and $g_{\boldsymbol{\theta}}(\mathcal{Z})$ based on samples. Second, also due to the invertibility of the generator, one can compute and monitor the distribution of the data in the latent space, $g_{\boldsymbol{\theta}}^{-1}(\mathcal{X})$ directly, and have a direct impact on the quality of the approximation on the $\mathcal{X}$ space. Ensuring sufficient similarity of this distribution and the latent distribution is vital to obtain meaningful samples. Third, it is possible to regularize the CNF problem using techniques from optimal transport (OT), which allows one to leverage theoretical results and accelerate the accuracy and efficiency of training algorithms. Fourth, OT-regularized CNFs can leverage recent advances made toward efficiently solving high-dimensional optimal transport problems using neural networks [51, 50, 43, 15, 36].

NF and CNF approaches' key limitation is their underlying assumptions, which are rarely satisfied in practice. Their applicability is limited to cases in which the intrinsic dimensionality of the dataset equals $n$, and there is a smooth and invertible transformation. In practice, one may experiment with NFs and CNFs even if it is unclear whether or not these assumptions, which are almost impossible to verify, hold. Suboptimal results could be due to inaccurate training or inadequate modeling but also could indicate the violation of one or both assumptions and motivate the use of VAE or GAN techniques.

**Variational Autoencoders** The crucial advantage of VAEs over NFs and CNFs is their ability to handle non-invertible generators and arbitrary dimension of the latent space. In the Bayesian setting, the training objective provides a lower bound on the likelihood that becomes tighter when the approximate posterior converges to the posterior implied by the generator. The training objective includes a reconstruction error, which can provide useful information about the latent space dimension. For example, if we observe a large error for expressive models, we may conclude that the data set's intrinsic dimension is larger than the latent space dimension. In our experience, the VAE training is more involved than that of NFs and CNFs due in part to the necessity to train a second network that parameterizes the approximate posterior.

Compared to GANs, which can handle the same class of generators, we found the training problem in VAEs to be less complicated. One reason for this is that VAE training requires minimizing a loss function and not solving a saddle point problem. Another reason is that we can use the understanding of the latent space to monitor the model's effectiveness; for example, by computing the reconstruction loss and the similarity of the samples from the approximate posterior to the latent distribution.

Our discussion also identified the different choices of sample distributions during the training phase and generation phase as one disadvantage of VAEs. During training, the latent variables are sampled

from the approximate posterior and not from the latent distribution. Even though the objective function penalizes the KL divergence between these distributions, in our example, the samples are generally not normally distributed; see Figure 8. This means that the generator may receive inputs it was not trained on during the sampling phase, which may lead to undesired effects. Since such generalization can generally not be expected from machine learning models such as neural networks, the lack of control of the latent samples remains a concern. It is important to note that it is possible to improve VAEs by modeling more complicated approximate posteriors, for example, using normalizing flows in the latent space [20].

**Generative Adversarial Networks** The training of GANs does not rely on estimates of the likelihood or latent variable. Instead, the training objective compares samples provided by the generator to those from the dataset without any correspondence. To this end, GANs introduce a second neural network, the discriminator, which we can construct in different ways to mimic binary classification or transport-based metrics. Despite considerable mathematical challenges, the popularity of GANs has been increasing dramatically in recent years. One reason for the surge in interest is their ability to produce visually indistinguishable samples from real data points and can be of higher quality than those predicted by generators trained using the VAE approach; see, for example, Figure 9.

The most apparent disadvantage of GANs is the difficulty of the training problem, which involves a saddle point problem and not, like in CNFs and VAEs, a minimization problem. Without theoretical advances that help guide the choice of hyperparameters, training GANs is likely to remain more of an art than a science. Along these lines, experimental evidence suggests that most GAN approaches can, after successful and cumbersome hyperparameter tuning, achieve similar results with respect to existing metrics [33]. The nonlinearity of the problem can cause various failure modes, including mode collapse or diverging iterations. In our experiments, we found that the performance is highly dependent on choosing the right hyperparameters such as batch size, learning rates, regularization parameters, and the architectures of the networks. In practice, this requires a repeated solution of a very costly training process. Therefore, we expect the computational costs of training a GAN in most cases to be considerably larger than training CNFs or VAEs. We found that initializing the generator using the weights from VAE training helped improve the training. Recent works that propose to base the training on variational inequalities [16, 13] promise a more reliable solution, but we have not included these in our experiments.

In our example, the transport-based Wasserstein GAN (WGAN) performed slightly better than the GAN based on binary classification. The theory of WGAN also has several key advantages, for example, reduced risk of mode collapse. However, there is a significant challenge of approximating the Wasserstein distance in high-dimensions efficiently using small batches. The formulation considered here requires optimizing a scalar-valued neural network subject to a constraint on its Lipschitz continuity and constant. This is a non-trivial endeavor, and developing rigorous methods that enforce this constraint could provide relevant improvements.

# 7 Outlook

As advances in machine learning and particularly deep learning enable the training of more powerful generative models, many remaining questions and challenges will almost surely lead to continued activity in deep generative modeling. In the following, we seek to identify some directions for future work related to but slightly beyond the topics covered by our paper.

At the core of deep generative modeling is the requirement to reliably and efficiently compare complicated, high-dimensional probability distributions. This has been a core problem of statistics for decades (if not longer), and bringing recent advances to bear in generative modeling is a fruitful direction of future research. Closing the gap between theory and practice is critical to improve the reliability of DGM training and reduce the immense computational costs. This paper has demonstrated the sampling problem in VAEs and enforcing the Lipschitz constraint in WGAN training.

While most existing DMG approaches use black-box neural networks as generators, there is a lack of models for incorporating domain-specific knowledge. This is a significant limitation, for example, in scientific use cases.

# Acknowledgments

This work was supported in part by NSF award DMS 1751636, AFOSR Grants 20RT0237, and US DOE Office of Advanced Scientific Computing Research Field Work Proposal 20-023231. We thank Elizabeth Newman, Malvern Madondo, and Tom O'Leary-Roseberry for proofreading an earlier version of this manuscript and providing many helpful comments.


# References


[1] L. Ardizzone, J. Kruse, S. Wirkert, D. Rahner, E. W. Pellegrini, R. S. Klessen, L. Maier-Hein, C. Rother, and U. Köthe. Analyzing Inverse Problems with Invertible Neural Networks. In *International Conference on Learning Representations*, 2018.

[2] M. Arjovsky and L. Bottou. Towards Principled Methods for Training Generative Adversarial Networks. *arXiv:1701.04862*, Jan. 2017.

[3] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein GAN. *arXiv:1701.07875*, Jan. 2017.

[4] A. S. Ashukha. Real NVP PyTorch. `https://github.com/senya-ashukha/real-nvp-pytorch`. Accessed: 2020-12-30.

[5] J.-D. Benamou and Y. Brenier. A computational fluid mechanics solution to the monge-kantorovich mass transfer problem. *Numerische Mathematik*, 84(3):375–393, 2000.

[6] L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[7] J. Brehmer, F. Kling, I. Espejo, and K. Cranmer. Madminer: Machine learning-based inference for particle physics. *Computing and Software for Big Science*, 4(1):1–25, 2020.

[8] A. Brock, J. Donahue, and K. Simonyan. Large Scale GAN Training for High Fidelity Natural Image Synthesis. *arXiv:1809.11096*, Sept. 2018.

[9] G. Carleo, I. Cirac, K. Cranmer, L. Daudet, M. Schuld, N. Tishby, L. Vogt-Maranto, and L. Zdeborová. Machine learning and the physical sciences. *arXiv:1903.10563*, (4):2773, Mar. 2019.

[10] D. Chu, I. Demir, K. Eichensehr, J. G. Foster, M. L. Green, K. Lerman, F. Menczer, C. O'Connor, E. Parson, and L. Ruthotto. White paper: Deep fakery—an action plan. Technical report, IPAM, 2020.

[11] L. Dinh, D. Krueger, and Y. Bengio. NICE: Non-linear Independent Components Estimation. *arXiv:1410.8516*, Oct. 2014.

[12] L. Dinh, J. Sohl-Dickstein, and S. Bengio. Density estimation using Real NVP. *arXiv:1605.08803*, May 2016.

[13] C. D. Enrich, S. Jelassi, C. Domingo-Enrich, D. Scieur, A. Mensch, and J. Bruna. Extragradient with player sampling for faster Nash equilibrium finding. *arXiv:1905.12363*, May 2019.

[14] L. C. Evans. Partial differential equations and monge-kantorovich mass transfer. *Current developments in mathematics*, 1997(1):65–126, 1997.

[15] C. Finlay, J.-H. Jacobsen, L. Nurbekyan, and A. Oberman. How to Train Your Neural ODE: the World of Jacobian and Kinetic Regularization. In *International Conference on Machine Learning*, pages 3154–3164. PMLR, Nov. 2020.

[16] G. Gidel, H. Berard, G. Vignoud, P. Vincent, and S. Lacoste-Julien. A Variational Inequality Perspective on Generative Adversarial Networks. *arXiv:1802.10551*, Feb. 2018.

[17] I. Goodfellow. NIPS 2016 Tutorial: Generative Adversarial Networks. *arXiv:1701.00160*, Dec. 2016.

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Nov. 2016.

[19] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. volume 27, pages 2672–2680, 2014.

[20] W. Grathwohl, R. T. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud. Ffjord: Free-form continuous dynamics for scalable reversible generative models. In *International Conference on Learning Representations*, 2018.

[21] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville. Improved Training of Wasserstein GANs. *Advances in neural information processing systems*, 30:5767–5777, 2017.

[22] E. Haber and R. Horesh. A Multilevel Method for the Solution of Time Dependent Optimal Transport. *Numerical Mathematics: Theory, Methods and Applications*, 8(01):97–111, Mar. 2015.

[23] P. Hagemann and S. Neumayer. Stabilizing invertible neural networks using mixture models. *Inverse Problems*, 02 2021.

[24] C. F. Higham and D. J. Higham. Deep learning: An introduction for applied mathematicians. *SIAM Review*, 61(4):860–891, 2019.

[25] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *36th International Conference on Machine Learning*, pages 448–456, Feb. 2015.

[26] T. Karras, S. Laine, and T. Aila. A Style-Based Generator Architecture for Generative Adversarial Networks. *CVPR*, pages 4401–4410, 2019.

[27] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv:1412.6980*, Dec. 2014.

[28] D. P. Kingma, T. Salimans, R. Jozefowicz, X. Chen, I. Sutskever, and M. Welling. Improving Variational Inference with Inverse Autoregressive Flow. *arXiv:1606.04934*, June 2016.

[29] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. *arXiv:1312.6114*, Dec. 2013.

[30] D. P. Kingma, M. Welling, et al. An introduction to variational autoencoders. *Foundations and Trends® in Machine Learning*, 12(4):307–392, 2019.

[31] Y. LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[32] J. Lin, K. Lensink, and E. Haber. Fluid Flow Mass Transport for Generative Networks. *arXiv:1910.01694*, Oct. 2019.

[33] M. Lucic, K. Kurach, M. Michalski, S. Gelly, and O. Bousquet. Are GANs Created Equal? A Large-Scale Study. *Advances in neural information processing systems*, 31:700–709, 2018.

[34] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida. Spectral Normalization for Generative Adversarial Networks. *arXiv:1802.05957*, Feb. 2018.

[35] F. Noé, S. Olsson, J. Köhler, and H. Wu. Boltzmann generators: Sampling equilibrium states of many-body systems with deep learning. *Science*, 365(6457), 2019.

[36] D. Onken, S. Wu Fung, X. Li, and L. Ruthotto. Ot-flow: Fast and accurate continuous normalizing flows via optimal transport. In *35th Conference on AAAI*, 2021.

[37] G. Papamakarios, T. Pavlakou, and I. Murray. Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pages 2338–2347, 2017.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[39] G. Peyré, M. Cuturi, et al. Computational optimal transport: With applications to data science. *Foundations and Trends® in Machine Learning*, 11(5-6):355–607, 2019.

[40] A. Radford, L. Metz, and S. Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv:1511.06434*, pages 1–16, 2015.

[41] D. Rezende and S. Mohamed. Variational inference with normalizing flows. In *International Conference on Machine Learning*, pages 1530–1538, 2015.

[42] D. J. Rezende, S. Mohamed, and D. Wierstra. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. *arXiv:1401.4082*, Jan. 2014.

[43] L. Ruthotto, S. J. Osher, W. Li, L. Nurbekyan, and S. Wu Fung. A machine learning framework for solving high-dimensional mean field game and mean field control problems. *Proceedings of the National Academy of Sciences*, 117(17):9183–9193, 2020.

[44] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen. Improved techniques for training gans. pages 2234–2242, 2016.

[45] SmartGeometry at UCL. CreativeAI: Deep Learning for Graphics Tutorial Code. `https://github.com/smartgeometry-ucl/dl4g`. Accessed: 2020-12-30.

[46] J. Stanczuk, C. Etmann, L. M. Kreusser, and C.-B. Schönlieb. Wasserstein gans work because they fail (to approximate the wasserstein distance), 2021.

[47] G. J. Székely and M. L. Rizzo. Testing for equal distributions in high dimension. *InterStat*, 5(16):1249–1272, 2004.

[48] T. Tieleman and G. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[49] C. Villani. *Topics in Optimal Transportation*. American Mathematical Soc., 2003.

[50] L. Yang and G. E. Karniadakis. Potential Flow Generator with $L_2$ Optimal Transport Regularity for Generative Models. *arXiv:1908.11462*, Aug. 2019.

[51] L. Zhang, W. E, and L. Wang. Monge-Ampère Flow for Generative Modeling. *arXiv:1809.10188*, Sept. 2018.