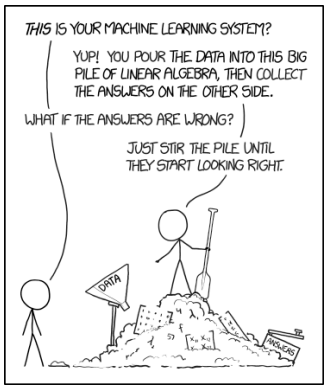
TÉCNICO LISBOA



Variational Mixture of Normalizing Flows

Guilherme P. Grijó Pires

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Mário Alexandre Teles de Figueiredo

Examination Committee

Chairperson: Prof. Full Name
Supervisor: Prof. Mário Alexandre Teles de Figueiredo
Member of the Committee: Prof. Full Name 3

Month Year

What I cannot create, I do not understand.

- Richard Feynman

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Nos últimos anos, os modelos generativos profundos, tais como redes generativas adversariais ([Goodfellow et al., 2014]) e auto-codificadores variacionais ([Diederik P Kingma and Welling, 2014]), e as suas variantes, têm vindo a ser amplamente adoptados para a tarefa de modelação de distribuições de dados. Apesar da qualidade excepcional das amostras sintécticas que estes modelos são capazes de produzir, as distribuições das mesmas são aprendidas *implicitamente*, no sentido em que não é possível aceder às funções de densidade de probabilidade que os modelos induzem. Isto torna-os inadequados para tarefas que requeiram, por exemplo, avaliar novos exemplos de dados com as distribuições aprendidas. Os *normalizing flows* ultrapassam esta limitação através da fórmula de mudança de variável para distribuições de probabilidade, e da utilização de transformações desenhadas para ter Jacobianas tratáveis e computáveis de forma viável. Apesar da sua flexibilidade, esta *framework* carecia (até à publicação de contribuições recentes: [Izmailov et al., 2019], [Dinh, Sohl-Dickstein, et al., 2019]) de uma forma de introduzir estrutura discreta (como a que se pode encontrar em modelos de mistura) nos modelos que permite construir, de forma não-supervisionada. Este trabalho pretende ser um passo nessa direcção, utilizando *normalizing flows* como componentes num modelo de mistura, e descrevendo um procedimento para o treino desse modelo. Este procedimento é baseado em inferência variacional e usa um posterior variacional parameterizado por uma rede neuronal. Como se tornará evidente, este modelo adequa-se naturalmente às tarefas de estimação de densidades (multimodais), aprendizagem semi-supervisionada, e aglomeração de dados. O modelo proposto é avaliado em dois conjuntos de dados sintécticos, e um conjunto de dados real.

Palavras-chave: Modelos generativos *deep*, normalizing flows, inferência variacional, modelação probabilística, aprendizagem automática.

Abstract

In the past few years, deep generative models, such as generative adversarial networks ([Goodfellow et al., 2014]) and variational autoencoders ([Diederik P Kingma and Welling, 2014]), and their variants, have seen wide adoption for the task of modelling the distribution of data. In spite of the outstanding sample qualities achieved with these methods, they model the target distributions *implicitly*, since the probability density functions induced by them are not accessible. This renders them unfit for tasks that require, for example, scoring new instances of data with the learned distributions. Normalizing flows overcome this limitation by leveraging the change-of-variables formula for probability density functions, and by using transformations designed to have tractable and cheaply computable Jacobians. Although flexible, this framework lacked (until the publication of recent work - [Izmailov et al., 2019], [Dinh, Sohl-Dickstein, et al., 2019]), a way to introduce discrete structure (such as the one found in mixture models) in the models it allows to construct, in an unsupervised sceario. The present work attempts to take a step in this direction by using normalizing flows as components in a mixture model, and devising a training procedure for such a model. This procedure is based on variational inference, and uses a variational posterior parameterized by a neural network. As will become clear, this model naturally lends itself to (multimodal) density estimation, semi-supervised learning, and clustering. The proposed model is evaluated on two synthetic datasets, as well as a real-world dataset.

Keywords: Deep generative models, normalizing flows, variational inference, probabilistic modelling, machine learning

Contents

List of Tables

List of Figures

List of Acronyms

ELBO	Evidence Lower Bound
EM	Expectation-Maximization
GAN	Generative Adversarial Network
GMM	Gaussian Mixture Model
MAF	Masked Autoregressive Flow
MAP	Maximum a-posteriori
MDL	Minimum Description Length
MLE	Maximum Likelihood Estimation
NF	Normalizing Flow
PReLU	Parameterized Rectified Linear Unit
RealNVP	Real Non Volume-Preserving
ReLU	Rectified Linear Unit
VAE	Variational Auto-Encoder
VMoNF	Variational Mixture of Normalizing Flows

Chapter 1

Introduction

1.1 Motivation and Related Work

Generative models based on neural networks - variational autoencoders (VAEs), generative adversarial networks (GANs), normalizing flows and their variations - have experienced increased interest and progress in their capabilities. VAEs ([Diederik P Kingma and Welling, 2014]) work by leveraging the reparameterization trick to optimize a variational posterior parameterized by a neural network, jointly with the generative model per se - it too a neural network, which takes samples from a latent as its input space and *decodes* them into the observation space. GANs also work by jointly optimizing two neural networks: a *generator*, which learns to produce realistic samples in order to "fool" the second network - the *discriminator* - which learns to distinguish samples produced by the generator from samples taken from data. GANs learn by having the generator and discriminator "compete". Both VAEs and GANs learn *implicit* distributions of the data, in the sense that - if training is successful - one can sample from the learned model, but there's no way to compute the likelihood of the learned distribution. Normalizing flows differ from VAEs and GANs in that they allow learning *explicit* distributions of the data[1]. Thus, normalizing flows lend themselves to the task of density estimation.

Less (although some) attention has been given to the extension of these types of models with discrete structure, such as the one found in mixture models. Exploiting such structure, while still being able to benefit from the expressiveness of neural generative models - specifically, normalizing flows - is the goal of this work. Concretely, this works explores a framework to learn a mixture of normalizing flows. In practice, a neural network classifier is learned jointly with the mixture components. Doing so will naturally produce an approach which lends itself not only to density estimation, but also clustering - since the classifier can be used to assign points to clusters - and semi-supervised learning, where available labels can be used to refine the classifier and selectively train the mixture components.

The work presented here intersects several active directions of research. In the sense of combining deep neural networks with probabilistic modelling, particularly with the goal of endowing simple probabilistic graphical models with more expressiveness, [Johnson et al., 2016] and [Lin, Khan, and

[1] In fact, recent work ([Grover, Dhar, and Ermon, 2018]) combines the training framework of GANs with the use of normalizing flows, so as to obtain a generator for which it is possible to compute likelihoods.

Hubacher, 2018] propose a framework to use neural network parameterized likelihoods, composed with latent probabilistic graphical models. Still in line with this topic, but with an approach more focused towards clustering and semi-supervised learning, [Dilokthanakul et al., 2016] proposes a VAE-inspired model, where the prior is a Gaussian mixture. [Xie, Girshick, and Farhadi, 2016] describes an unsupervised method for clustering using deep neural networks, which is a task that can also be fulfilled by the model presented in this work.

The following are brief descriptions of the two works that are most related to the present work.

[Dinh, Sohl-Dickstein, et al., 2019], similarly to this work, tries to reconcile normalizing flows with a multimodal/discrete structure. It does so by partitioning the latent space into disjoint subsets, using a Mixture Model where each component has non-zero weight exclusively within its respective subset. Then, using a set identification function and a piecewise invertible function, a variation of the change-of-variable formula is devised.

[Izmailov et al., 2019] also exploits multimodal structure while using normalizing flows for expressiveness. However, while the present work relies on a variational posterior parameterized by a neural network and learns K flows (one for each mixture component), [Izmailov et al., 2019] resorts to the use of a latent mixture of Gaussians as the base distribution for its flow model, and it learns a single flow.

1.2 Objectives

The objectives of the present work are

- designing a mixture of normalizing flows, with a tractable learning procedure;
- a proof-of-concept implementation to demonstrate the capabilities of such model, namely in the tasks of:
  - Density estimation;
  - Clustering;
  - Semi-supervised learning.

1.3 Thesis Outline

In Chapter 2, the concepts on probabilistic modelling needed for the remainder of the work are introduced. In Chapter 3, the framework and theoretical background of normalizing flows is presented. Chapter 4 describes in detail the model proposed in the present work, and Chapter 5 contains empirical results and their interpretation.

1.4 Notation

The notation used throughout this work is as follows:

- Scalars and vectors are lower-case letters. To differentiate between them, vectors will be present in bold. E.g.: x is a scalar $\mathbf{z}$ is a vector.

- Upper-case letters are matrices.
- For vectors, $\boldsymbol{x}_{a:b}$ denotes the a-th to the b-th elements of $\boldsymbol{x}$.
- For distributions, subscript notation will only be used when the distribution isn't clear from context.
- The operator $\odot$ denotes the element-wise product.
- The letter x is preferred for observations.
- The letter z is preferred for latent variables.
- The letter $\boldsymbol{\theta}$ is preferred for parameter vectors.
- A function g of $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$, which is parameterized by a parameter vector $\boldsymbol{\theta}$ is written as $g(\boldsymbol{x}; \boldsymbol{\theta})$, when the dependence on $\boldsymbol{\theta}$ is to be made explicit.

Chapter 2

Probabilistic Modelling

2.1 Introduction

The goal of probabilistic modelling is a to leverage probability distributions and random variables to posit, test, and refine hypothesis about the behaviour of systems that include stochastic components. Given observations of a system, the task of probabilistic modelling normally boils down to finding a probability distribution which:

- is consistent with the observed data;
- is consistent with **new**, previously unobserved data, originated in the same system.

This probability distribution is commonly called the *model*. A good model will be a good *emulator* of the true generative process that originated the observed data. In informal terms, this can be summarized as:

$$\text{data} \sim p(\text{data}|\text{hypothesis}^*), \tag{2.1}$$

where hypothesis* is the optimal hypothesis. Via Bayes' Law, we can write:

$$p(\text{hypothesis}|\text{data}) = \frac{p(\text{data}|\text{hypothesis})p(\text{hypothesis})}{p(\text{data})}. \tag{2.2}$$

In practice, a modeller will search for an hypothesis that maximizes (some form, or approximation of) this expression. For simpler problems, this search happens in closed-form, i.e., there is an expression to compute the optimal hypothesis, given data. However, for most real-world problems, there is no closed-form solution, and the modeller has to resort to algorithms and approximations, and will only be able to find **local** optima for the above expression, in most cases.

It's also worth noting that there are effectively infinite candidate distributions - each one an hypothesis for how the system at hand generates data. It is common to make use of domain knowledge and assume the true system has a certain intrinsic structure and form, and to use these assumptions to constrain the space of candidate hypothesis. Assumptions about structure usually translate to conditional independence claims between some or all of the observed variables; assumptions about form translate into the

use of specific parametric families to govern some or all of the observed variables. These assumptions are commonly connected between themselves (for example, when conjugate likelihood-prior pairs are used). When parametric forms are used, an hypothesis is uniquely defined by the set of parameters it requires - commonly denoted θ[1].

2.2 Model Complexity

Intuitively, the dimension of θ is deeply connected with the expressiveness of the distribution. In practice, this translates to the observation that if we make the model[2] expressive enough, it can fit the observed data arbitrarily well. Naïvely, this would be a desirable characteristic to exploit - it's always possible to increase the likelihood by adding parameters to the model. However, increasing model complexity normally comes at the expense of generalization. This phenomenon is commonly referred to as *overfitting*, and there are several angles from which to explain it and interpret it. Namely:

- The classical perspective is that of the *bias-variance tradeoff*. To understand this, consider the concept of an hypothesis class - a set of hypotheses in which, via some procedure, the modeller will search for an hypothesis that is consistent with the observed data, and is expected to generalize to unseen data. Said procedure is what is normally referred to as *fitting* the model to the data. In the case of parametric models, the set of models of a given parametric form, with a parameter-vector of a certain fixed dimension, is an example of an Hypothesis Class. Intuitively, a more complex hypothesis class is more likely to contain the true hypothesis (or a good approximation to it). However, the more complex the hypothesis class, the larger the search space - the higher the number of candidate hypothesis. In this sense, an increase in the size of the search space often translates into an increase of the sensitivity to the problem variables (in the case of learning and inference, this means sensitivity to initialization and to the data used to fit). Conversely, a simpler model will constitute a smaller search-space, hence the search procedure will be less sensitive to initalization and problem variables. However, the true hypothesis (or a good approximation to it) is less likely to be contained in it - precisely because it is a smaller hypothesis class. The bias-variance tradeoff is a summary of these observations: a highly complex model is potentially able to achieve a low expected error on observed data (low bias), but will tend to be extremely sensible to small variations on its input (high variance). Conversely, a simpler model will be more robust to variations on its input (low variance), but won't have the same modelling capacity and will produce a larger expected error (high bias).[3]

[1] For the type of models and problems dealt with in this work, I will assume θ is finite, but it's worth noting that there are models for which the dimension of θ *grows* with the dataset size. These are called non-parametric models. They come with their own advantages and disadvantages, which are out of the scope of this work.

[2] Throughout this work I will be using the words *model* and *distribution* almost interchangeably, making it clear when context isn't enough.

[3] The number of parameters is far from being the best measure of complexity of a model. Nevertheless, it is a good proxy to compare model complexity between models of the same parametric family. However, recent work by Belkin et al. (Belkin et al., 2018) shows that modern machine learning contexts, in which the number of parameters is far larger than in classical settings, have to be understood under a measure of model complexity different from the traditional ones. This is because it is now common practice to fit highly overparameterized models to a point of interpolation (close to zero training error), still being able to achieve good generalization - which evidently contradicts the classical view.

- Andrey Kolmogorov's and Gregory Chaitin's ideas on algorithmic information theory (AIT) Chaitin, 2016, and Kolmogorov complexity are another useful lens through which to regard this question. Consider that data are measurements of phenomena. Modelling is concerned with finding the laws that explain/govern these phenomena. Intuitively, if the laws are as complex as the data they intend to explain, they aren't explaining anything. AIT formalizes this notion by borrowing the concept of *program* to define the generative process by which observed data comes to existence. The complexity of a dataset is then easy to define: it is the size of the **smallest**[4] program that generates the observed data. The appropriate unit with which to measure the size of a program - and, as we have now seen, the complexity of a dataset - is bits[5]. The parallel between these ideas and the question of overfitting is thus easy to make: a program (or a model and its parameter vector) is useful if it *compresses* the data, intuitively because to do so it leverages the patterns therein, which are the object of interest in the modelling task.

Both of these lines of reasoning make clear that there is a certain balance in complexity that a good model has to achieve: it should be parsimonious enough that it won't overfit, but flexible enough that it is able to properly explain the observed data. There are strategies to make this mathematically objective. Some of those methods are the Bayesian Information Criterion, the Akaike Information Criterion and the Minimum Description Length ([Lanterman, 2001]).

2.3 MAP and ML Estimation

Once the parametric form of the model is defined, the task at hand becomes the discovery of the parameter vector $\boldsymbol{\theta}$ that best explains the observed data, within the defined parametric family. The naïve (but often the only possible) approach is to maximize what is called the likelihood function. That is, given observation x:

$$\hat{\boldsymbol{\theta}}_{ML} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, \mathcal{L}(\boldsymbol{\theta}), \tag{2.3}$$

where $\mathcal{L}(\boldsymbol{\theta}) = p(x|\boldsymbol{\theta})$. This approach is called *maximum likelihood estimation*. Note that $\mathcal{L}$ is a function of $\boldsymbol{\theta}$. Depending on the model, finding $\boldsymbol{\theta}^*$ - the optimum - can be as simple as computing an analytical expression, or as difficult as using gradient-based methods to optimize a non-convex objective. In the latter case, local optima are usually the best one can expect to obtain.

Another approach, called *maximum a posteriori*, works by retrieving the mode (or one mode, if there are several) of the posterior distribution of the parameter-vector, given by:

$$\hat{\boldsymbol{\theta}}_{MAP} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, p(\boldsymbol{\theta}|x), \tag{2.4}$$

where $p(\boldsymbol{\theta}|x) = \frac{p(x|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(x)}$.

It's easy to see that these two approaches are intimately related. MAP differs from ML by the fact

[4] Note the emphasis on "smallest" - this is because any program can be made arbitrarily redundant, and thus arbitrarily large.
[5] Or the basic unit of memory of the computer where the data generating program would run

that it employs the prior $p(\boldsymbol{\theta})$ to give different weights to different hypothesis (i.e., different instances of $\boldsymbol{\theta}$). This is useful if there is domain knowledge available that can be encoded in the prior. It is worth noting that ML is a special case of MAP, when the prior is uniform.

2.4 Structure and Latent Variables

In some cases, one might want to leverage some available domain knowledge. This often translates into assuming that there is some latent structure in the data. This structure is commonly encoded into latent variables and their influence over the observable variables.

In this scenario, we become interested in the distribution given by $p(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta_x}, \boldsymbol{\theta_z})$, where z is the latent variable, $\boldsymbol{\theta_x}$ is the parameter vector for the distribution of $\boldsymbol{x} \in \boldsymbol{\mathcal{X}}$, and $\boldsymbol{\theta_z}$ is the parameter vector for the distribution of $\boldsymbol{z} \in \boldsymbol{\mathcal{Z}}$.

For structure and latent variables to be useful, it is common to make the additional assumption that we have the ability of factorizing their joint distribution in ways that make it tractable. If we have a dataset $\boldsymbol{X}$, with N i.i.d. samples $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}, ..., \boldsymbol{x_N}$ and N latent variables $\boldsymbol{z_1}, \boldsymbol{z_2}, \boldsymbol{z_3}, ..., \boldsymbol{z_N}$, one common factorization is:

$$p(\boldsymbol{X}, \boldsymbol{Z}, \boldsymbol{\theta}) = \Big(\prod_{i=1}^{N} p(\boldsymbol{x_i}|\boldsymbol{z_i}, \boldsymbol{\theta_x}) p(\boldsymbol{z_i}|\boldsymbol{\theta_z}) \Big) p(\boldsymbol{\theta_x}) p(\boldsymbol{\theta_z}). \tag{2.5}$$

It's also possible that the samples in $\boldsymbol{X}$ have some sort of causal relation, for instance if they occur ordered in time. In this case, they are not i.i.d. One way to encode this assumption is to posit an *autoregressive* model, i.e., a model in which a random variable depends on the realizations of the variables that come before it. If each random variable depends solely on the random variable that precedes it, this is called a Markov model. A common variation of the Markov model is the hidden Markov Model, where the autoregressive part of the model is present only in the latent variables:

$$p(\boldsymbol{X}, \boldsymbol{Z}, \boldsymbol{\theta}) = p(\boldsymbol{z_1}) \Big(\prod_{i=2}^{N} p(\boldsymbol{x_i}|\boldsymbol{z_i}, \boldsymbol{\theta_x}) p(\boldsymbol{z_i}|\boldsymbol{z_{i-1}}, \boldsymbol{\theta_z}) \Big) p(\boldsymbol{\theta_x}) p(\boldsymbol{\theta_z}). \tag{2.6}$$

These are merely (classical) examples of models with different structure assumptions encoded into them. Normally, if the structure has a certain regularity, it's possible to exploit it to obtain tractable (approximate) inference and estimation methods. This notion is explored in the following section, for a subset of the family of models described by 2.5.

2.5 Mixture Models and the EM Algorithm

Mixture models are a subset of the structure "family" described in 2.5, and they have a central role in this work. In a Mixture Model there is a discrete latent variable z_i which selects one of K components

from which an observation $\boldsymbol{x_i}$ will be sampled. This can be summarized as:

$$z_i \sim p(z_i|\boldsymbol{\pi}) \tag{2.7}$$

$$x_i \sim p(\boldsymbol{x_i}|z_i) \tag{2.8}$$

The probability of $\boldsymbol{x_i}$ being sampled from component k (that is, the proability of $z_i = k$) is commonly referred to as the *weight* of component k. It is common to assume that all of the K components are part of the same parametric family. In that case, we can rewrite the above as:

$$z_i \sim p(z_i|\boldsymbol{\pi}) \tag{2.9}$$

$$x_i \sim p(\boldsymbol{x_i}|\boldsymbol{\theta}_{z_i}), \tag{2.10}$$

where it is made evident that the discrete variable z_i is selecting the **parameter vector** to be used for sample x_i. The most discussed mixture model is the Gaussian Mixture Model, in which the K components of the model are Gaussians.

The EM Algorithm

The Expectation-Maximization algorithm is the most commonly used algorithm to fit a mixture model[6], i.e. to estimate its parameters. The starting point of EM is the realisation that if all variables were observed, it would be easy to apply ML or MAP estimation for the parameters. Given that, the algorithm can be generally described as an alternation between two steps, described next.

- **E-step**: infer the most probable values of the unobserved variables. (In the case of mixture models, this corresponds to inferring the discrete variables that select the component from which each observed data point was sampled. This can be roughly thought of as a cluster assignment).
- **M-step**: given the observed variables and the values inferred on the previous step, optimize the model parameters. (In the case of mixture models, this correponds to inferring the parameters of each component, and its weight).

A more rigorous description of the procedure follows. Consider the expression

$$\ell_c(\boldsymbol{\theta}) = \sum_{i=1}^{N} \log p(\boldsymbol{x_i}, z_i|\boldsymbol{\theta}), \tag{2.11}$$

which is the so called complete data log-likelihood. Like the likelihood function, it is a function of $\boldsymbol{\theta}$, which is easy to compute, given all the $\boldsymbol{x_i}$ and z_i. However, the z_i aren't observed. To overcome this, let us define the expected complete data log likelihood:

$$Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t-1)}) = \mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\ell_c(\boldsymbol{\theta})|X, \boldsymbol{\theta}^{(t-1)}], \tag{2.12}$$

where $\boldsymbol{\theta}^{(t)}$ represents the value of $\boldsymbol{\theta}$ at time step t of the fitting procedure. Note that the expectation is

[6] Although it can be used to fit other types of models.

w.r.t. Z, given X and $\boldsymbol{\theta}^{(t-1)}$. It is the expected value of $\ell_c(\boldsymbol{\theta})$, given the parameter values obtained at the previous step of the the algorithm. Depending on the nature of Z, this expectation can be obtained either in closed form or approximated, for instance via samples of z_i (if a sampling procedure is available).

In the case of Mixture Models, where the z_i are instances of a categorical random variable, the expression for $Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t-1)})$ can be made simpler as such:

$$
\begin{aligned}
Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t-1)}) &= \mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\ell_c(\boldsymbol{\theta})|X, Z, \boldsymbol{\theta}^{(t-1)}] && (2.13)\\
&= \mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\sum_{i=1}^{N} \log p(x_i, z_i|\boldsymbol{\theta})] && (2.14)\\
&= \mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\sum_{i=1}^{N} \log \prod_{k=1}^{K} \left(\pi_k p(x_i|\boldsymbol{\theta}_k)\right)^{\mathbb{I}(z_i=k)}] && (2.15)\\
&= \mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\sum_{i=1}^{N} \sum_{k=1}^{K} \log \left(\left(\pi_k p(x_i|\boldsymbol{\theta}_k)\right)^{\mathbb{I}(z_i=k)}\right)] && (2.16)\\
&= \mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\sum_{i=1}^{N} \sum_{k=1}^{K} \mathbb{I}(z_i = k) \log \left(\pi_k p(x_i|\boldsymbol{\theta}_k)\right)] && (2.17)\\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} \underbrace{\mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\mathbb{I}(z_i = k)]}_{\text{Let this quantity be represented by } r_{ik}} \log \left(\pi_k p(x_i|\boldsymbol{\theta}_k)\right) && (2.18)\\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \log \left(\pi_k p(x_i|\boldsymbol{\theta}_k)\right) && (2.19)\\
&= \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \log \pi_k + \sum_{i=1}^{N} \sum_{k=1}^{K} r_{ik} \log p(x_i|\boldsymbol{\theta}_k) && (2.20)\\
& && (2.21)
\end{aligned}
$$

The quantity defined above as r_{ik} is referred to as the *responsability*. It is trivial to arrive at a closed-form expression for it, given the value of $\boldsymbol{\theta}$ arrived at on the previous iteration, i.e. $\boldsymbol{\theta}^{(t-1)}$:

$$
\begin{aligned}
r_{ik} &= \mathbb{E}_{Z|X,\boldsymbol{\theta}^{(t-1)}}[\mathbb{I}(z_i = k)] && (2.22)\\
&= p(z_i = k|x_i\ ; \boldsymbol{\theta}^{(t-1)}) && (2.23)\\
&= \frac{p(z_i = k, x_i\ ; \boldsymbol{\theta}^{(t-1)})}{p(x_i\ ; \boldsymbol{\theta}^{(t-1)})} && (2.24)\\
&= \frac{p(z_i = k, x_i\ ; \boldsymbol{\theta}^{(t-1)})}{\sum_{k'} p(z_i = k', x_i\ ; \boldsymbol{\theta}^{(t-1)})} && (2.25)\\
&= \frac{\pi_k p(x_i|\boldsymbol{\theta}_k^{(t-1)})}{\sum_{k'} \pi_{k'} p(x_i|\boldsymbol{\theta}_{k'}^{(t-1)})} && (2.26)
\end{aligned}
$$

The EM algorithm for mixture models alternates between the following two steps:

- **E-step**: Compute the responsabilies r_{ik} for each x_i.

- **M-step**: Given r_{ik}, solve the following optimization problem:

$$\boldsymbol{\theta}^{(t)} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, Q(\boldsymbol{\theta}, \boldsymbol{\theta}^{(t-1)}) \underbrace{+ \log p(\boldsymbol{\theta})}_{\text{Optional, if we want to do MAP estimation}}, \tag{2.27}$$

  which can have a closed-form solution in some of the simplest mixture models, like Gaussian mixture models, but can require a gradient-based optimization procedure for more flexible models.

2.6 Approximate Inference

Take the expression $p(\boldsymbol{x}, \boldsymbol{z}, \boldsymbol{\theta})$. For simplicity, let us consider θ as part of the latent variables $\boldsymbol{z}$. This means that the model is simply written as the joint distribution: $p(\boldsymbol{x}, \boldsymbol{z})$. To perform inference[7] about $\boldsymbol{z}$, conditioned on $\boldsymbol{x}$, it is necessary to find the posterior distribution of $\boldsymbol{z}$, given $\boldsymbol{x}$, i.e. $p(\boldsymbol{z}|\boldsymbol{x})$. Using Bayes' Law:

$$p(\boldsymbol{z}|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{p(\boldsymbol{x})} \tag{2.28}$$

$$= \frac{p(\boldsymbol{x}|\boldsymbol{z})p(\boldsymbol{z})}{\int p(\boldsymbol{x}|\boldsymbol{z'})p(\boldsymbol{z'})d\boldsymbol{z'}} \tag{2.29}$$

For the vast majority of cases, the integral on the denominator will be intractable. To overcome this difficulty, it is common to resort to two families of methods: Monte-Carlo methods - which are out of the scope of this work - and variational methods.

2.6.1 Variational Methods

Variational methods (essentially) work by turning the problem of integration into one of optimization. They propose a family of parametric distributions, and then optimize the parameters so as to minimize the "distance" between the approximate (normally called "variational") distribution and the distribution of interest. There are two ways to derive the most commonly used objective function for this problem, which will be detailed in the two following subsections.

Kullback-Leibler Divergence

The Kullback-Leibler divergence is a measure of the dissimilarity between two probability distributions p, and q, given by

$$KL(q||p) = \int q \log \frac{q}{p}, \tag{2.30}$$

[7] If we hadn't collapsed $\boldsymbol{\theta}$ into $\boldsymbol{z}$ and were instead handling it separatly, we would call **inference** to the task of finding $\boldsymbol{z}$ and **learning** to the task of finding $\boldsymbol{\theta}$

in the case of continuous variables.

In the setting of inference, p is the posterior $p(\boldsymbol{z}|\boldsymbol{x})$ and q is a distribution in some parametric family, with parameters ϕ, i.e., $q(\boldsymbol{z};\phi)$. However, it is clear that we can't compute the Kullback-Leibler directly, because it requires the knowledge of both distributions, and finding $p(\boldsymbol{z}|\boldsymbol{x})$ is precisely the task at hand. Let us expand the KL divergence expression:

$$
\begin{aligned}
KL(q||p) &= \int q(\boldsymbol{z})(\log q(\boldsymbol{x}) - \log p(\boldsymbol{z}|\boldsymbol{x}))dz && (2.31)\\
&= \int q(\boldsymbol{z})(\log q(\boldsymbol{z}) - (\log p(\boldsymbol{x},\boldsymbol{z}) - \log p(\boldsymbol{x})))d\boldsymbol{z} && (2.32)\\
&= \mathbb{E}_q[\log q(\boldsymbol{z})] - \mathbb{E}_q[\log p(\boldsymbol{x},\boldsymbol{z})] + \mathbb{E}_q[\log p(\boldsymbol{x})] && (2.33)\\
&= \mathbb{E}_q[\log q(\boldsymbol{z})] - \mathbb{E}_q[\log p(\boldsymbol{x},\boldsymbol{z})] + \log p(\boldsymbol{x}) && (2.34)\\
& && (2.35)
\end{aligned}
$$

The last term is constant w.r.t $q(\boldsymbol{z})$. Consequently, for a fixed $p(\boldsymbol{x})$, minimizing the KL divergence is equivalent to minimizing

$$\mathbb{E}_q[\log q(\boldsymbol{z})] - \mathbb{E}_q[\log p(\boldsymbol{x},\boldsymbol{z})], \tag{2.36}$$

which is equivalent to maximizing

$$\mathbb{E}_q[\log p(\boldsymbol{x},\boldsymbol{z})] - \mathbb{E}_q[\log q(\boldsymbol{z})]. \tag{2.37}$$

This quantity is commonly refered to as ELBO - Evidence Lower BOund. It can be rewritten as:

$$
\begin{aligned}
ELBO(q) &= \mathbb{E}_q[\log p(\boldsymbol{x},\boldsymbol{z})] - \mathbb{E}_q[\log q(\boldsymbol{z})] && (2.38)\\
&= \mathbb{E}_q[\log p(\boldsymbol{x}|\boldsymbol{z})] + \mathbb{E}_q[\log p(\boldsymbol{z})] - \mathbb{E}_q[\log q(\boldsymbol{z})] && (2.39)
\end{aligned}
$$

In this form, each term of the ELBO has an easily interpretable role:

- $\mathbb{E}_q[\log p(\boldsymbol{x}|\boldsymbol{z})]$ tries to maximize the conditional likelihood of $\boldsymbol{x}$. That can be seen as assigning high probability mass to values of $\boldsymbol{z}$ that *explain* $\boldsymbol{x}$ well.
- $\mathbb{E}_q[\log p(\boldsymbol{z})]$ is the symmetric of the crossentropy between $q(\boldsymbol{z})$ and $p(\boldsymbol{z})$. Maximizing this quantity is equivalent of minimizing that crossentropy. This can be regarded as a regularizer that discourages $q(\boldsymbol{z})$ from being too different from the prior $p(\boldsymbol{z})$.
- $-\mathbb{E}_q[\log q(\boldsymbol{z})]$ is the entropy of $q(\boldsymbol{z})$. Maximizing this term incentivizes the probability mass of $q(\boldsymbol{z})$ to be spread out: another form of regularization.

A lower bound on $\log p(\boldsymbol{x})$

Another way of approaching the intractable posterior is to start by stating that our inherent goal is to maximize $p(\boldsymbol{x})$, or equivalently $\log p(\boldsymbol{x})$. Given that, consider the following chain of equalities and inequalities,

$$\log p(\boldsymbol{x}) = \log \int p(\boldsymbol{x}, \boldsymbol{z}) d\boldsymbol{z} \tag{2.40}$$

$$= \log \int q(\boldsymbol{z}) \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})} d\boldsymbol{z} \tag{2.41}$$

$$= \log \mathbb{E}_q[\frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})}] \tag{2.42}$$

$$\geq \mathbb{E}_q[\log \frac{p(\boldsymbol{x}, \boldsymbol{z})}{q(\boldsymbol{z})}] \tag{2.43}$$

$$\geq \mathbb{E}_q[\log p(\boldsymbol{x}, \boldsymbol{z})] - \mathbb{E}_q[q(\boldsymbol{z})] \tag{2.44}$$

To understand the inequality between (2.42) and (2.43), consider Jensen's inequality, given (in one of its forms) by:

$$\phi(\mathbb{E}[X]) \leq \mathbb{E}[\phi(X)], \tag{2.45}$$

where $\phi(.)$ is a convex function. If $\xi(.)$ is a concave function, then $-\xi(.)$ is a convex function, and we obtain the reverse inequality (substituting $\phi(.)$ with $-\xi(.)$ in the inequality in (2.45)): $\xi(\mathbb{E}[X]) \geq \mathbb{E}[\xi(X)]$. This form is the most useful for us, since $\log$ is a concave function. Using this, the step between (2.42) and (2.43) is made obvious.

Note that the right-hand side of 2.44 is the same quantity we arrived at in 2.37, and that it is a lower-bound on the quantity we want to maximize, and so we want to maximize it. It's worth noting that when $q(\boldsymbol{z}) = p(\boldsymbol{z}|\boldsymbol{x})$, the bound is tight.

Chapter 3

Normalizing Flows

3.1 Introduction

The best known and studied probability distributions, which are analitically manageable, are rarely expressive enough for real-world complex datasets, such as images or signals. However, they have properties that make them amenable to work with, for instance: tractable parameter estimation, closed-form likelihood functions, and simple sampling procedures.

As described in the previous chapter, one way to obtain more expressive models is to assume the existence of latent variables, leverage certain factorization structures, and to use well-known distributions for the individual factors of the product that constitutes the model's joint distribution. By using these structures and choosing specific combinations of distributions (namely, conjugate prior-likelihood pairs), these models are able to stay tractable - normally via bespoke estimation/inference/learning algorithms.

Another approach to obtaining expressive probabilistic models is to apply transformations to a simple distribution, and use the *change of variables* formula to compute probabilities in the transformed space. This is the basis of *normalizing flows*, an approach proposed by Rezende and Mohamed in [Rezende and Mohamed, 2015], and which has since evolved and developed into the basis of multiple state-of-the-art techniques for density modelling and estimation ([Durk P Kingma and Dhariwal, 2018], [Dinh, Sohl-Dickstein, and Bengio, 2017], [De Cao, Titov, and Aziz, 2019], [Papamakarios, Pavlakou, and Murray, 2017]).

3.2 Change of Variables

Given a random variable $\boldsymbol{z}$, with probability density function f_Z, and a bijective and continuous function g, the probability density function f_X of the random variable $\boldsymbol{x} = g(\boldsymbol{z})$ is given by

$$f_X(\boldsymbol{x}) = f_Z(g^{-1}(\boldsymbol{x}))\left|\det\left(\frac{d}{d\boldsymbol{x}}g^{-1}(\boldsymbol{x})\right)\right|. \tag{3.1}$$

If g is a transformation parameterized by a parameter vector $\boldsymbol{\theta}$, this expression can be optimized w.r.t. $\boldsymbol{\theta}$, with the goal of making it approximate some arbitrary distribution. For this to be feasible, the following have to be easily computable:

- f_Z - the starting distribution's probability density function (also called *base density*). It is assumed that there is a closed-form expression to compute this. In practice, this is typically one of the basic distributions (Gaussian, Uniform, etc.)
- $\det\left(\frac{d}{d\boldsymbol{x}}g^{-1}(\boldsymbol{x};\boldsymbol{\theta})\right)$ - the determinant of the Jacobian matrix of g^{-1}. For most transformations, this is not "cheap" to compute.
- The gradient of $\det\left(\frac{d}{d\boldsymbol{x}}g^{-1}(\boldsymbol{x};\boldsymbol{\theta})\right)$ w.r.t. $\boldsymbol{\theta}$. This is crucial for gradient-based optimization of $\boldsymbol{\theta}$ to be feasible. For most cases, this is not easily computable.

As will become clear, the crux of the normalizing flows framework is to find transformations that are expressive enough, and for which the determinants of their Jacobian matrices, as well as the gradients of those determinants are "cheap" to compute.

3.3 Normalizing Flows

Let us have L transformations h_ℓ, for $\ell = 0, 1, ..., L$ that fulfill the three requirements listed above. Let each of those transformations be parameterizable by a parameter vector $\boldsymbol{\theta}_\ell$, for $\ell = 0, 1, ..., L$[1]. Let $\boldsymbol{z_\ell} = h_{\ell-1} \circ h_{\ell-2} \circ ... \circ h_0(\boldsymbol{z_0})$, where $\boldsymbol{z_0}$ is sampled from f_Z, the base density. Furthermore, let g be the composition of the L transformations. Applying the change of variables formula to

$$\boldsymbol{z_0} \sim f_Z \tag{3.2}$$

$$\boldsymbol{x} = h_{L-1} \circ h_{L-2} \circ ... \circ h_0(\boldsymbol{z_0}) \tag{3.3}$$

then

$$f_X(\boldsymbol{x}) = f_Z(g^{-1}(\boldsymbol{x}))\left|\det\left(\frac{d}{d\boldsymbol{x}}g^{-1}(\boldsymbol{x})\right)\right| \tag{3.4}$$

$$= f_Z(g^{-1}(\boldsymbol{x}))\prod_{\ell=0}^{L-1}\left|\det\left(\frac{d}{d\boldsymbol{z_{\ell+1}}}h_\ell^{-1}(\boldsymbol{z_{\ell+1}})\right)\right| \tag{3.5}$$

$$= f_Z(g^{-1}(\boldsymbol{x}))\prod_{\ell=0}^{L-1}\left|\det\left(\frac{d}{d\boldsymbol{z_\ell}}h_\ell\left(h_\ell^{-1}(\boldsymbol{z_{\ell+1}})\right)\right)\right|^{-1} \tag{3.6}$$

Replacing $h_\ell^{-1}(\boldsymbol{z_{\ell+1}}) = \boldsymbol{z_\ell}$ in 3.6 leads to

$$f_X(\boldsymbol{x}) = f_Z(g^{-1}(\boldsymbol{x}))\prod_{\ell=0}^{L-1}\left|\det\left(\frac{d}{d\boldsymbol{z_\ell}}h_\ell\left(\boldsymbol{z_\ell}\right)\right)\right|^{-1}; \tag{3.7}$$

[1] The dependence on the parameter vectors will be implicit from here on

taking the logarithm,

$$\log f_X(\boldsymbol{x}) = \log f_Z(g^{-1}(\boldsymbol{x})) - \sum_{\ell=0}^{L-1} \log \left| \det \left(\frac{d}{d\boldsymbol{z_\ell}} h_\ell(\boldsymbol{z_\ell}) \right) \right| \tag{3.8}$$

Depending on the task, one might prefer to replace the second term in 3.8 with a sum of log-absolute-determinants of the Jacobians of the inverse transformations. This switch would imply replacing the minus sign before the sum with a plus sign:

$$\log f_X(\boldsymbol{x}) = \log f_Z(g^{-1}(\boldsymbol{x})) + \sum_{\ell=0}^{L-1} \log \left| \det \left(\frac{d}{d\boldsymbol{z_{\ell+1}}} h_\ell^{-1}(\boldsymbol{z_{\ell+1}}) \right) \right| \tag{3.9}$$

We started by assuming that the transformations h_ℓ fulfilled the requirements listed in section 3.2. For that reason, it is clear that the above expression is a feasible objective for gradient-based optimization[2]. Sampling from the resulting distribution is simply achieved by sampling from the base distribution and applying the chain of transformations. Because of this, normalizing flows lend themselves to be used as flexible variational posteriors, in variational inference settings, as well as density estimators.

3.4 Examples of transformations

3.4.1 Affine Transformation

An affine transformation is arguably the simplest choice. This transformation can stretch, sheer, shrink, rotate, and translate the space. It is simply achieved by the multiplication by a matrix A and summation of a bias vector $\boldsymbol{b}$:

$$\boldsymbol{z} \sim p(\boldsymbol{z}) \tag{3.10}$$

$$\boldsymbol{x} = A\boldsymbol{z} + \boldsymbol{b} \tag{3.11}$$

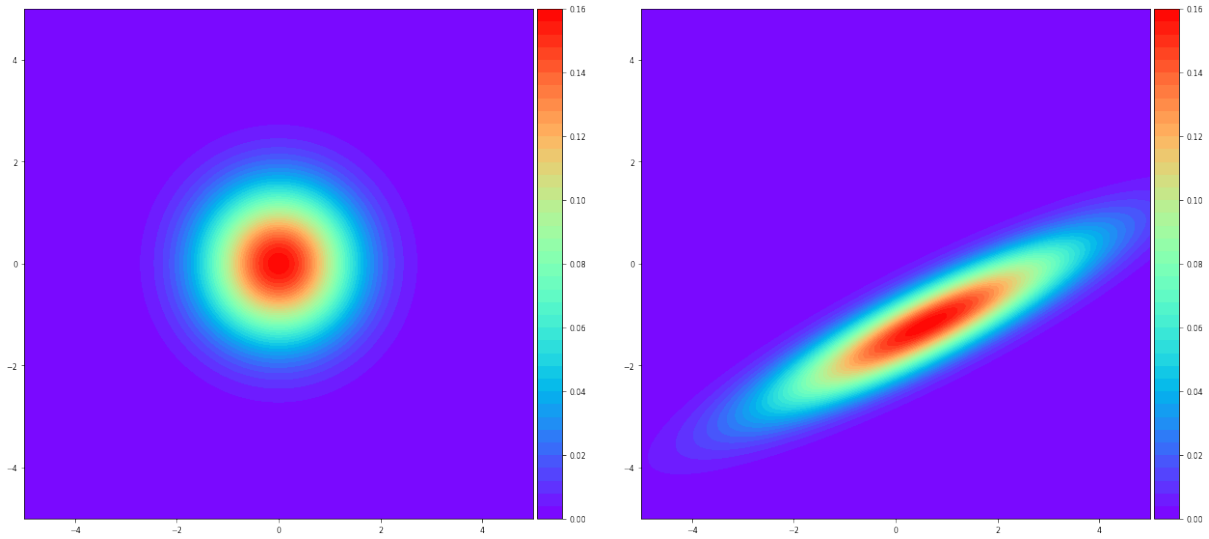
The determinant of the Jacobian of this transformation is simply the determinant of the matrix A. However, in general, computing the determinant of a $N \times N$ matrix has $\mathcal{O}(N^3)$. For that reason, it is common to use matrices with a certain structure which makes their determinants easier to compute. For instance, if A is triangular, its determinant is simply the product of its diagonal's elements. The downside of using matrices that are constrained to a certain structure is that they correspond to less flexible transformations.

It is possible, however, to design affine transformations whose Jacobian determinants are of $\mathcal{O}(N)$ complexity and that are more expressive than simple triangular matrices. In [Durk P Kingma and Dhariwal, 2018], one such transformation is proposed. It constrains the matrix A to be decomposable as $A = PL\big(U + diag(\boldsymbol{s})\big)$, where $diag(\boldsymbol{s})$ is a diagonal matrix whose diagonal's elements are the values of vector $\boldsymbol{s}$. The following additional constrains are in place:

[2] In practice, this is carried out by leveraging modern automatic differentiation and optimization frameworks.

- P is a permutation matrix
- L is a lower triangular matrix, with ones in the diagonal
- U is an upper triangular matrix, with zeros in the diagonal

Given these constraints, the determinant of the matrix A is simply the product of the elements of s.



(a) (b)

Figure 3.1: (a) Density of a Gaussian distribution with $\mu = [0, 0]$ and $\Sigma = I$ (b) Density of the distribution that results from applying some affine transformation to the Gaussian distribution in (a)

3.4.2 PReLU Transformation

Intuitively, introducing non-linearity endows normalizing flows with more flexibility to represent complex distributions. This can be done in similar fashion to the activation functions of neural networks. One example of that, is the parameterized rectified linear unit transformation. It is defined in the following manner, for a d-dimensional input:

$$f_i(z_i) = \begin{cases} z_i, & \text{if } z_i \geq 0 \\ \alpha z_i, & \text{otherwise} \end{cases} \tag{3.12}$$

$$f(\boldsymbol{z}) = [f_0(z_0), f_1(z_1), ..., f_d(z_d)] \tag{3.13}$$

Note that in order for the transformation to be invertible, it is necessary that $\alpha > 0$.

Let us define an auxiliary function $j(.)$ as

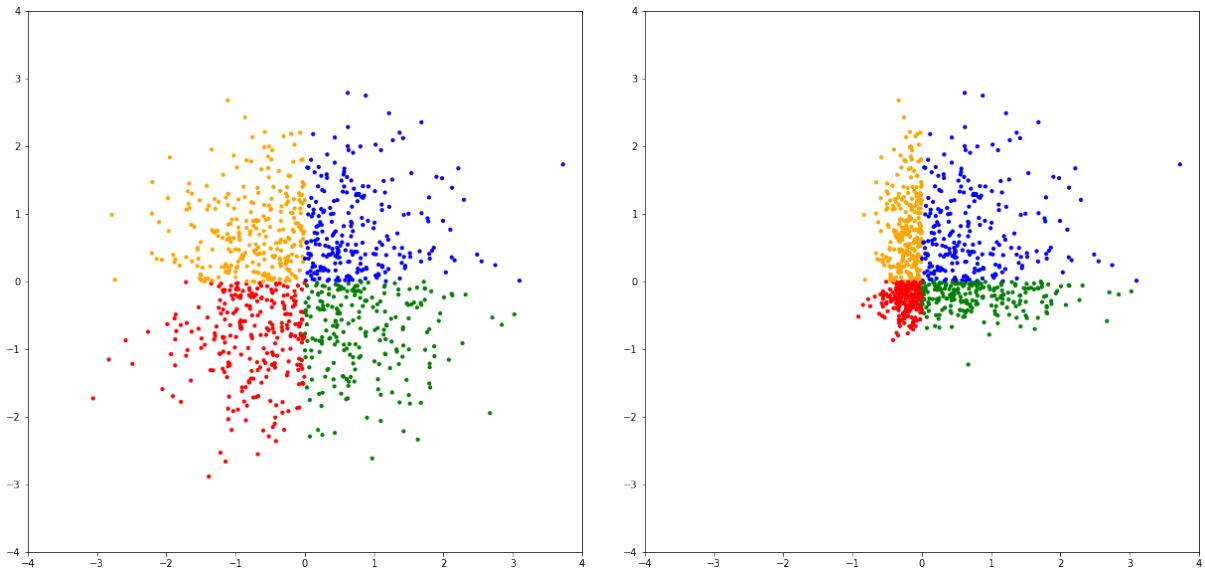
$$j(z_i) = \begin{cases} 1, & \text{if } z_i \geq 0 \\ \alpha, & \text{otherwise} \end{cases} \tag{3.14}$$

It's trivial to see that the Jacobian of the transformation is a diagonal matrix, whose diagonal elements

are $j(z_i)$:

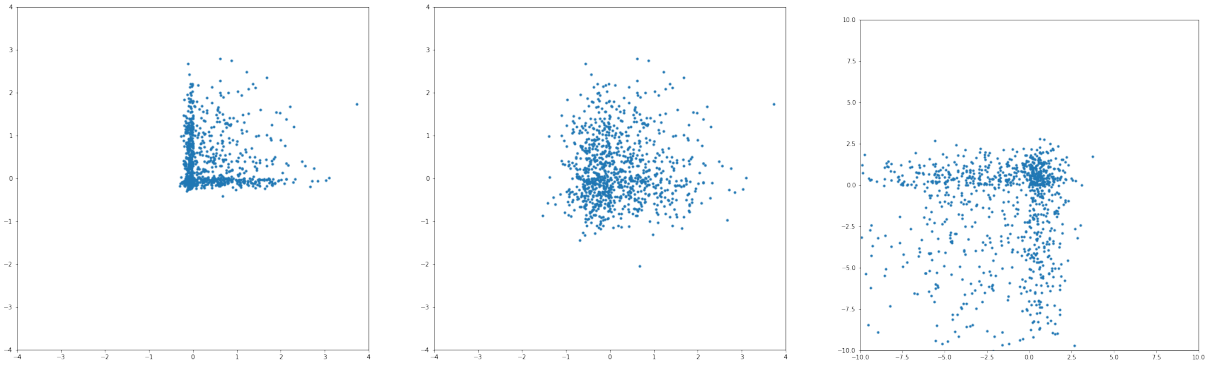
$$J(f(z)) = \begin{bmatrix} j(z_0) & & & \\ & j(z_1) & & \\ & & \ddots & \\ & & & j(z_d) \end{bmatrix} \tag{3.15}$$

With that, it is easy to arrive at the log-absolute-determinant of this transformation's Jacobian, which is given by $\sum_{i=0}^{d} \log \left| j(z_i) \right|$



(a) (b)

Figure 3.2: (a) Samples from of a Gaussian distribution with $\mu = [0, 0]$ and $\Sigma = I$. The samples are colored according to the quadrant they belong to. (b) Samples from the distribuion in a) transformed by a PReLU transformation.



(a) (b) (c)

Figure 3.3: Samples from a Gaussian with $\mu = [0, 0]$ and $\Sigma = I$, transformed by PReLU transformations with different α parameters. (a) $\alpha = 0.1$ (b) $\alpha = 0.5$ (c) $\alpha = 5$

3.4.3 Batch-Normalization Transformation

In [Dinh, Sohl-Dickstein, and Bengio, 2017], the authors propose a batch-normalization transformation, similar to the well-known batch-normalization layer normally used in neural networks. This transform simply applies a rescaling, given the batch mean $\tilde{\mu}$ and variance $\tilde{\sigma}^2$:

$$f(z) = \frac{z - \tilde{\mu}}{\sqrt{\tilde{\sigma}^2 + \epsilon}}, \tag{3.16}$$

where $\epsilon \ll 1$ is a term used to ensure that there never is a division by zero. This transformation's Jacobian is trivial:

$$\prod_i \frac{1}{\sqrt{\tilde{\sigma}_i^2 + \epsilon}}. \tag{3.17}$$

3.4.4 Affine Coupling Transformation

As mentioned previously, one of the active research challenges within the normalizing flows framework is the search and design of transformations that are expressive and whose Jacobians are not computationally heavy. One brilliant example of such transformations was proposed in [Dinh, Sohl-Dickstein, and Bengio, 2017], and is called affine coupling layer.

This transformation is characterized by two **arbitrary** functions $s(.)$ and $t(.)$, as well as a mask that splits an input $\boldsymbol{z}$ of dimension D into two parts, $\boldsymbol{z_1}$ and $\boldsymbol{z_2}$. In practice, $s(.)$ and $t(.)$ are neural networks, whose parameters are to be optimized so as to make the transformation approximate the desired output distribution. The outputs of $s(.)$ and $t(.)$ need to have the same dimension as $\boldsymbol{z_1}$. This should be taken into account when designing the mask and the functions $s(.)$ and $t(.)$. It is defined as:

$$\boldsymbol{x_1} = \boldsymbol{z_1} \odot \exp\big(s(\boldsymbol{z_2})\big) + t(\boldsymbol{z_2}) \tag{3.18}$$

$$\boldsymbol{x_2} = \boldsymbol{z_2} \tag{3.19}$$

To see why this transformation is suitable to being used within the framework of normalizing flows, let us derive its Jacobian.

- $\frac{\partial \boldsymbol{x_2}}{\partial \boldsymbol{z_2}} = I$ is trivial, because $\boldsymbol{x_2} = \boldsymbol{z_2}$.
- $\frac{\partial \boldsymbol{x_2}}{\partial \boldsymbol{z_1}}$ is a matrix of zeros, because $\boldsymbol{x_2}$ does not depend on $\boldsymbol{z_1}$.
- $\frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_1}}$ is a diagonal matrix, whose diagonal is simply given by $\exp\big(s(\boldsymbol{z_2})\big)$, since those values are constant w.r.t $\boldsymbol{z_1}$ and they are multiplying each element of $\boldsymbol{z_1}$.
- $\frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_2}}$ is not needed for our purposes, as will become clear ahead.

Writing the above in matrix form:

$$J_{f(z)} = \begin{bmatrix} \frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_1}} & \frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_2}} \\ \frac{\partial \boldsymbol{x_2}}{\partial \boldsymbol{z_1}} & \frac{\partial \boldsymbol{x_2}}{\partial \boldsymbol{z_2}} \end{bmatrix} \tag{3.20}$$

$$= \begin{bmatrix} diag\Big(\exp\big(s(\boldsymbol{z_2})\big)\Big) & \frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_2}} \\ \boldsymbol{0} & I \end{bmatrix} \tag{3.21}$$

shows that the Jacobian matrix is triangular. Its determinant - the only thing we need, in fact - is therefore easy to compute: it is simply the product of the diagonal elements. Moreover, part of the diagonal is simply composed of ones. The determinant, and the log-absolute-determinant become

$$det\big(J_{f(z)}\big) = \prod_i \exp\big(s(\boldsymbol{z_2}^{(i)})\big) \tag{3.22}$$

$$\log\Big|det\big(J_{f(z)}\big)\Big| = \sum_i s(\boldsymbol{z_2}^{(i)}), \tag{3.23}$$

where $\boldsymbol{z_2}^{(i)}$ is the i-th element of $\boldsymbol{z_2}$. Since a single affine coupling layer does not transform all of the elements in $\boldsymbol{z}$, in practice several layers are composed, and each layer's mask is changed so as to make all dimensions affect each other. This can be done, for instance, with a checkerboard pattern, which alternates for each layer. In the case of image inputs, the masks can operate at the channel level.

3.4.5 Masked Autoregressive Flows

Another ingenious architecture for normalizing flows has been proposed in [Papamakarios, Pavlakou, and Murray, 2017]. It is called Masked Autoregressive Flow (MAF). Let $\boldsymbol{z}$ be a sample from some base distribution, with dimension D. MAF transforms $\boldsymbol{z}$ into an observation $\boldsymbol{x}$, of the same dimension, in the following manner:

$$x_i = z_i \exp(\alpha_i) + \mu_i \tag{3.24}$$

$$\mu_i, \alpha_i = g(\boldsymbol{x_{1:i-1}}) \tag{3.25}$$

In the above expression g is some arbitrary function. The inverse transform of MAF is trivial, because, like the Affine Coupling Layer, MAF uses g to parameterize a shift, mu, and a log-scale, $alpha$, which translates to the fact that the function g itself doesn't need to be inverted:

$$z_i = (x_i - \mu_i) \exp(-\alpha_i) \tag{3.26}$$

Moreover, the autoregressive structure of the transformation constrains the Jacobian to be triangular, which renders the determinant effortless to compute:

$$\det\left(J_{f(\boldsymbol{z})}\right) = \prod_{i}^{D} \exp(\alpha_i) \tag{3.27}$$

$$\log\left|\det\left(J_{f(\boldsymbol{z})}\right)\right| = \sum_{i}^{D} \alpha_i \tag{3.28}$$

As stated above, the function g used to obtain μ_i and α_i can be arbitrary. However, in the original paper, the function proposed to that end is a Masked Autoencoder for Distribution Estimation (MADE), as described in [Germain et al., 2015].

Much like the partitioning in the Affine Coupling Layer, the assumption of autoregressiveness (and the ordering of the elements of $\boldsymbol{x}$ in which for which that assumption is held) carries an inductive bias with it. Again, like with the Affine Coupling Layer, this effect is minimized in practice by stacking layers with different element orderings.

3.5 Fitting Normalizing Flows

Generally speaking, normalizing flows can be used in one of two scenarios: (direct) density estimation, where the goal is to optimize the parameters so as to make the model approximate the distribution of some observed set of data; in a variational inference scenario, as way of having a flexible variational posterior (i.e. $q(\boldsymbol{z})$ in the terminology used in the previous chapter). The second scenario is out of the scope of this work.

The task of density estimation with normalizing flows reduces to finding the optimal parameters of a parametric model. As mentioned in section 2.3, there are two ways to go about estimating the parameters of a parametric model, given data: MLE and MAP. In the case of normalizing flows, MLE is the usual approach[3]. To fit a normalizing flow via MLE, a gradient based optimizer is used to minimize $\hat{\mathcal{L}}(\boldsymbol{\theta}) = -\mathbb{E}[\log p(\boldsymbol{x}|\boldsymbol{\theta})]$

[3] In theory it is possible to place a prior on the normalizing flow's parameters and do MAP estimation. To accomplish this, similar strategies to those used in Bayesian Neural Networks would have to be used.

Chapter 4

Variational Mixture of Normalizing Flows

4.1 Introduction

As mentioned in sections 2.4 and 2.5, the ability of leveraging domain knowledge to endow a probabilistic model with structure is often useful. The goal of this work is to devise a model that combines the flexibility of normalizing flows with the ability to exploit class-membership structure. This is achieved by learning a mixture of normalizing flows, via optimization of a variational objective, for which the variational posterior over the class-indexing latent variables is parameterized by a neural network. Intuitively, this neural network should learn to place similar instances of data in the same class, allowing each component of the mixture to be fitted to a cluster of data.

4.2 Model Definition

Let us define a mixture model as in section 2.5, where each of the K components is a normalizing flow. For simplicity, consider that all of the K normalizing flows have the same architecture[1], i.e., they are all composed of the same stack of transformations, but they each have their own parameters.

Additionally, let $q(z|\boldsymbol{x};\gamma)$ be a neural network with a softmax output, with parameters γ. This network will receive as input an instance from the data, and produce the probability of that instance belonging to each of the K classes.

Recall the evidence lower bound given in 2.37[2]:

$$\text{ELBO} = \mathbb{E}_q[\log p(\boldsymbol{x}, z)] - \mathbb{E}_q[\log q(z|\boldsymbol{x})]$$

[1] This is not a requirement, and in cases where we have classes with different levels of complexity, we can have components with different architectures. However, the training procedure does not guarantee that that the most flexible normalizing flow is "allocated" to the most complex cluster. This is definitely an interesting direction for future research.

[2] Here the dependence of q on x is made explicit

Let us rearrange it:

$$\text{ELBO} = \mathbb{E}_q[\log p(\boldsymbol{x}|z)] + \mathbb{E}_q[\log p(z)] - \mathbb{E}_q[\log q(z|\boldsymbol{x})] \tag{4.1}$$

$$= \mathbb{E}_q[\log p(\boldsymbol{x}|z) + \log p(z) - \log q(z|\boldsymbol{x})] \tag{4.2}$$

Since $q(z|\boldsymbol{x})$ is given by the forward-pass of a neural network, and is therefore straightforward to obtain, the expectation in 4.2 is given by computing the expression inside the expectation for each possible value of z, and summing the obtained values, weighed by the probabilities given by the variational posterior. Thus, the whole ELBO is easy to compute, provided that each of the terms inside the expectation is itself easy to compute. Let us consider each of those terms:

- $\log p(\boldsymbol{x}|z)$ is the log-likelihood of $\boldsymbol{x}$ under the normalizing flow indexed by z. It was shown in the previous chapter how to compute this.
- $\log p(z)$ is the log-prior of the component weights. For simplicity, let us assume this is set by the modeller. When nothing is known about the component weights, the best assumption is that they are uniform. Nevertheless, as will be shown empirically, this too can be optimized.
- $-\log q(z|\boldsymbol{x})$ is the negative logarithm of the output of the encoder.

For a better intuition about each of these terms, it is useful to review the last paragraph of the first subsection of Section 2.6.1.

Let us call this model Variational Mixture of Normalizing Flows (VMoNF). For an overview of the model, consider figures 4.1 and 4.2

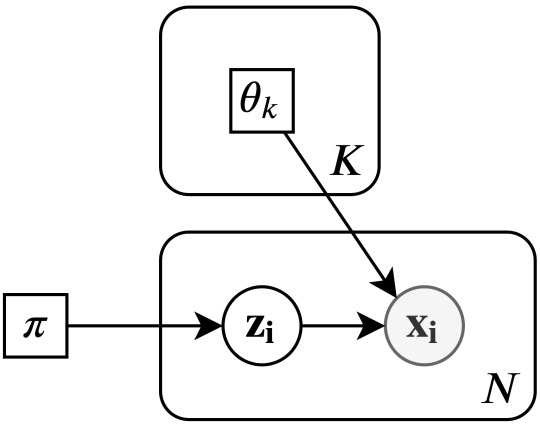


Figure 4.1: Plate diagram of a Mixture of K Normalizing Flows. $\boldsymbol{\theta}_k$ is the parameter vector of component k.

In a similar fashion to the Variational Auto-Encoder, proposed in [Diederik P Kingma and Welling, 2014], a VMoNF is fitted by jointly optimizing the parameters of the variational posterior $q(z|\boldsymbol{x};\boldsymbol{\gamma})$ and the parameters of the generative process $p(\boldsymbol{x}|z;\boldsymbol{\theta})$.

After training, the variational posterior can be directly used as a classifier for new data points.

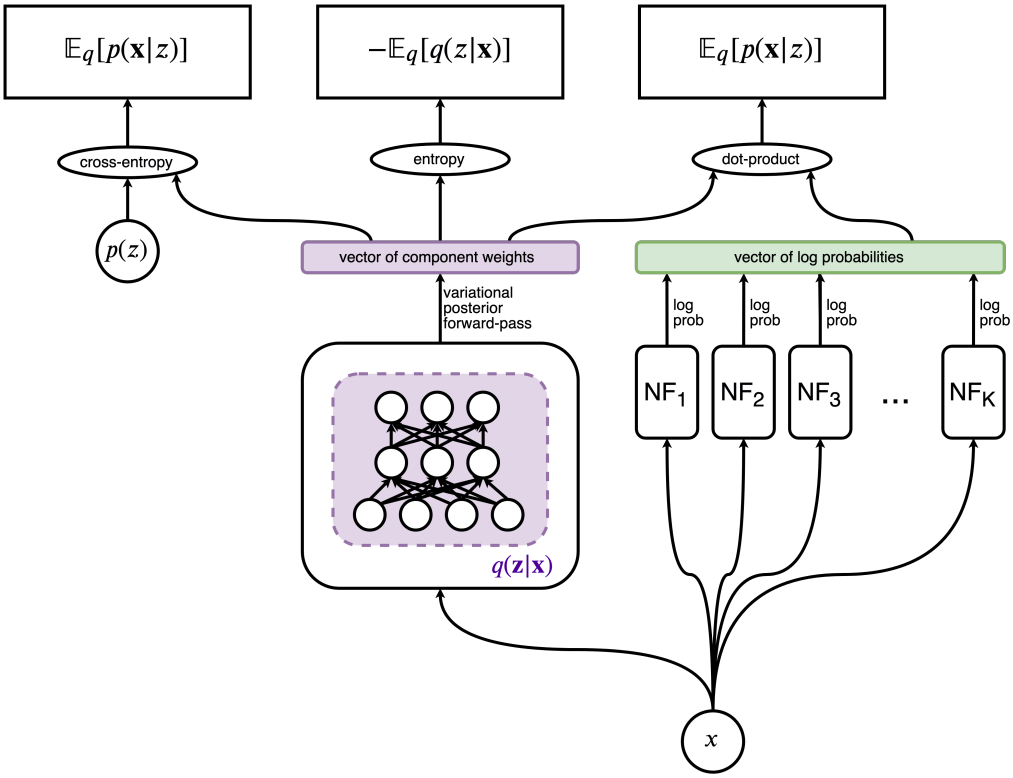


Figure 4.2: Overview of the training procedure.

4.3 Implementation

To implement and test the proposed model, Python was the chosen language. More specifically, this work heavily realies on the PyTorch ([Paszke et al., 2017]) package and framework for automatic differentiation. Moreover, the parameter optimization is done via stochastic optimization, namely using the optimizer proposed in [Diederick P Kingma and Ba, 2015] - Adam.

Figure 4.2 gives an overview of the training procedure:

1. The *log-probabilities* given by each component of the mixture are computed.
2. The values of the variational posterior probabilities for each component are computed.
3. With the results of the previous steps, all three terms of the ELBO are computable.
4. The ELBO and its gradients w.r.t the model parameters are computed and the parameters are updated.
5. Steps 1 to 4 are repeated until some stopping criterion is met.

Chapter 5

Experiments

In this chapter, the proposed model is applied to two synthetic datasets (Pinwheel and Two-circles) and one real-world dataset (MNIST). On one of the synthetic datasets, one shortcoming of the model is brought to attention, but is overcome in a semi-supervised setting. On the real-world dataset, the model's clustering capabilities are evaluted, as well as its capacity to model complex distributions.

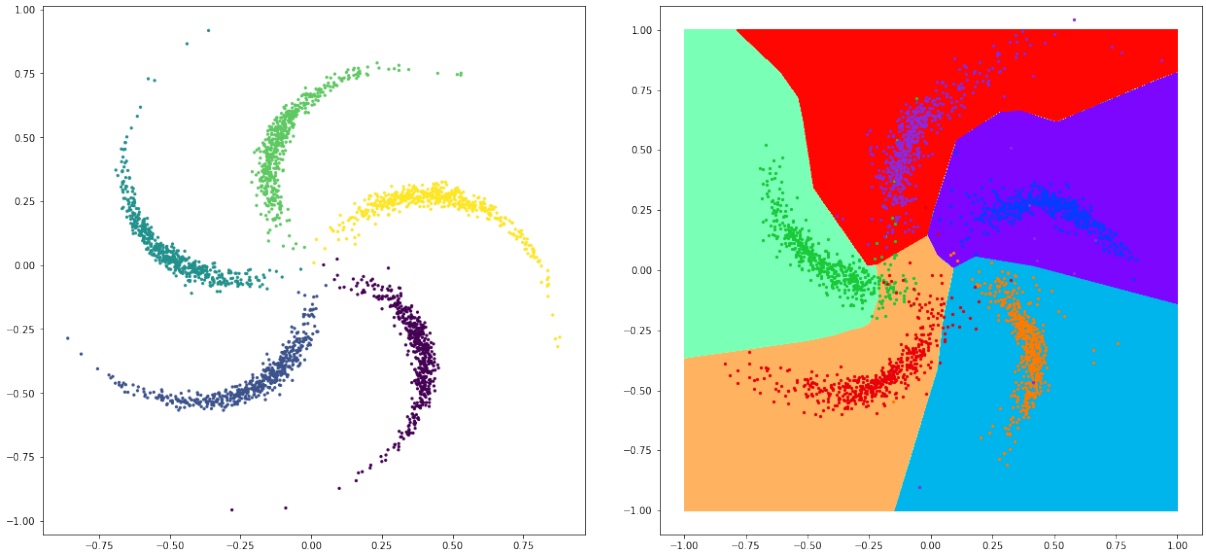
All experiments were conducted using RealNVP (the model proposed in [Dinh, Sohl-Dickstein, and Bengio, 2017]) as the normalizing flow model for the mixture components. Moreover, a technique inspired in [Zhang et al., 2017] was employed to improve training speed and quality of results. This consisted of dividing the inputs of the softmax layer in the variational posterior by a temperature value, t, which was made to follow an exponential decay schedule during training. Intuitively, this makes the variational posterior "more certain" as training moves on, while allowing all components to be generally exposed to the whole data, in the initial epochs. This incentivizes components to not be "subtrained" in the initial epochs and then prematurely discarded by the variational posterior.

5.1 Toy datasets

5.1.1 Pinwheel dataset

This dataset is constituted by five non-linear "wings". See figure 5.1 for the results of running the model on this dataset. As expected, the variational posterior has learned to partition the space so as to attribute each "wing" to a component of the mixture. This partitioning is imperfect in regions of space that have low probability for every component.

This experiment consisted of training on 2560 data points, 512 per class; using the Adam ([Diederick P Kingma and Ba, 2015]) optimizer, with a learning rate of 0.001, with a mini-batch size of 512, during 400 epochs. The variational posterior was parameterized by a multi-layer perceptron, with 1 hidden layer of dimension 3, and with a softmax output. Each component of the mixture was a RealNVP with 8 blocks, each block with multi-layer perceptrons, with 1 hidden layer of dimension 8, as the $s(.)$ and $t(.)$ functions of the affine coupling layers.



(a) (b)

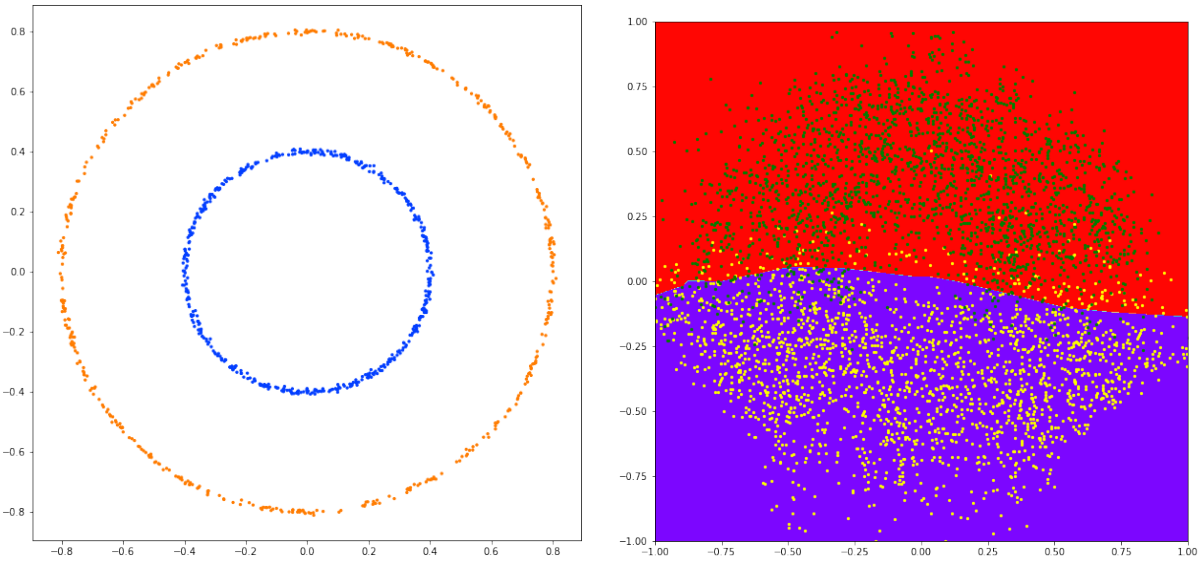
Figure 5.1: (a) Original dataset. (b) Samples from the learned model. Each dot is colored according to the component it was sampled from. The background colors denote the regions where each component has maximum probability assigned by the variational posterior. (Note that the background colors were chosen so as to not match the dot colors, otherwise it dots wouldn't be visible)

5.1.2 Two-circles dataset

This dataset consists of two concentric circles. The experiment on this dataset, visible on figure **??**, makes evident one shortcoming of this model: the way in which the variational posterior partitions space is not necessarily guided by the intrisic structure in the data. In the case of the two-circles dataset, it was found that the most common space partitioning induced by the model consisted simply of splitting space in half. However, in a semi-supervised setting, this behaviour can be corrected and the model successfully learns to separate the two circles, as is visible in figure 5.3. In this setting, the model was pretrained on the labeled instances for some epochs and then trained with the normal procedure. In the semi-supervised setting the model has the chance to refine both the variational posterior and each of the components, thus making better use of the unlabeled data in the unsupervised phase of the training. As is clearly visible in figure 5.3, the model struggles with learning full, closed, circles. This is because it is unable to "pierce a hole" in the base distribution, due to the nature of the transformations that are applicable. Thus, to model a circle, the model has to learn to stretch the blob formed by the base distribution, and "bend it over itself". This difficulty is also what keeps the model from learning a structurally interesting solution in the fully unsupervised case: it is easier to learn to distort space so as to learn a multimodal distribution that models half of the two circles.

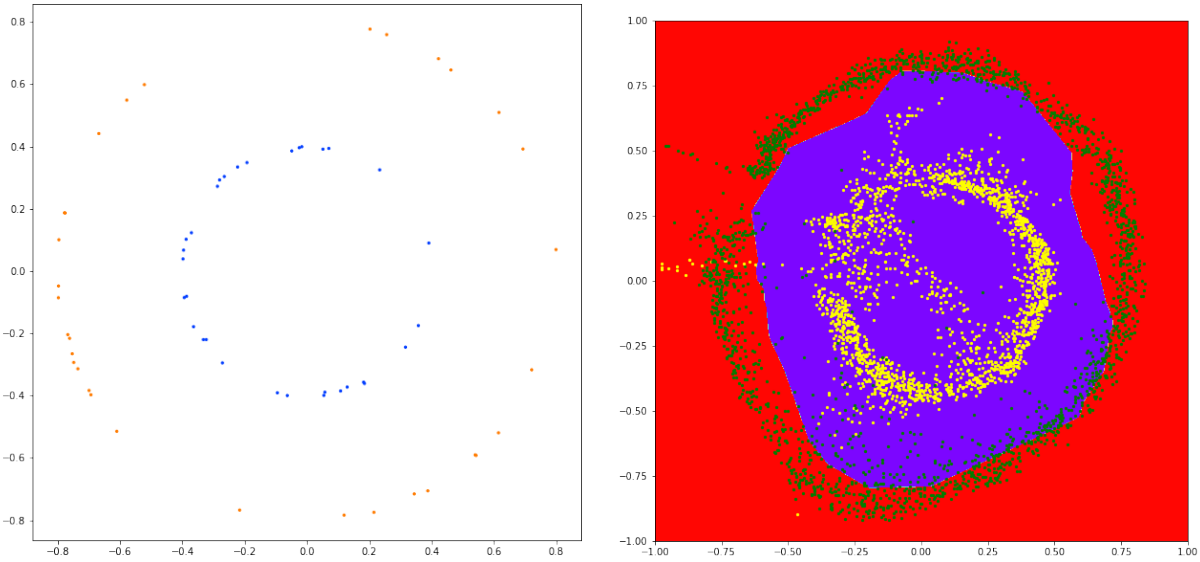
The unsupervised learning experiment consisted of training on 1024 datapoints, 512 per class; using the Adam ([Diederick P Kingma and Ba, 2015]) optimizer, with a learning rate of 0.001, with a mini-batch size of 128, during 500 epochs. The semi-supervised learning experiment consisted of training on 1024 unlabeled datapoints, 512 per class; and 32 labeled data points, 16 per class. The model was first pretrained during 300 epochs solely on the 32 labeled data points, using the labels to selectively optimize each component of the mixture, as well as to optimize the variational posterior by minimizing a

binary cross-entropy loss. After pretraining, the model was trained by interweaving supervised epochs - like in pretraining - with unsupervised epochs. Optimization was carried out using the Adam ([Diederick P Kingma and Ba, 2015]) optimizer, with a learning rate of 0.001, with a mini-batch size of 128, during 500 epochs. For both the unsupervised and the semi-supervised experiments, the neural network used to parameterize the variational posterior was a multi-layer perceptron, with 2 hidden layers of dimension 16, and with a softmax output. Each component of the mixture was a RealNVP with 10 blocks, each block with multi-layer perceptrons, with 1 hidden layer of dimension 8, as the $s(.)$ and $t(.)$ functions of the affine coupling layers.



(a) (b)

Figure 5.2: (a) Original dataset. (b) Samples from the learned model, without any labels. Coloring logic is the same as in 5.1. (c) Samples from semi-supervised scenario.



(a) (b)

Figure 5.3: (a) Labeled points used in semi-supervised scenario. (b) Samples from the model trained in the semi-supervised scenario.

5.2 Real-world dataset

In this section, the proposed model is evaluated in the well-known MNIST dataset([LeCun and Cortes, 2010]). This dataset consists of pixel-matrices of images of handwritten digits. The grids are of dimension 28 x 28, and were flattened to vectors of dimension 784 for training. For this experiment, only the images corresponding to the digits from 0 to 4 were considered. The normalizing flow model used for the components was a MAF, with 5 blocks, whose internal MADE layers had 1 hidden layer of dimension 200. The variational posterior was parameterized by a multi-layer perceptron, with 1 hidden layer of dimension 512. The model was trained for 100 epochs, with a mini-batch size of 100. The Adam optimizer was used, with a learning rate of 0.0001, and with a weight decay parameter of 0.000001. In figure **??**, samples from the components obtained after training can be seen. Moreover, a normalized contingency table is presented, where the performance of the variational posterior as a clustering function can be assessed. Note that the cluster indices induced by the model have no semantic meaning.

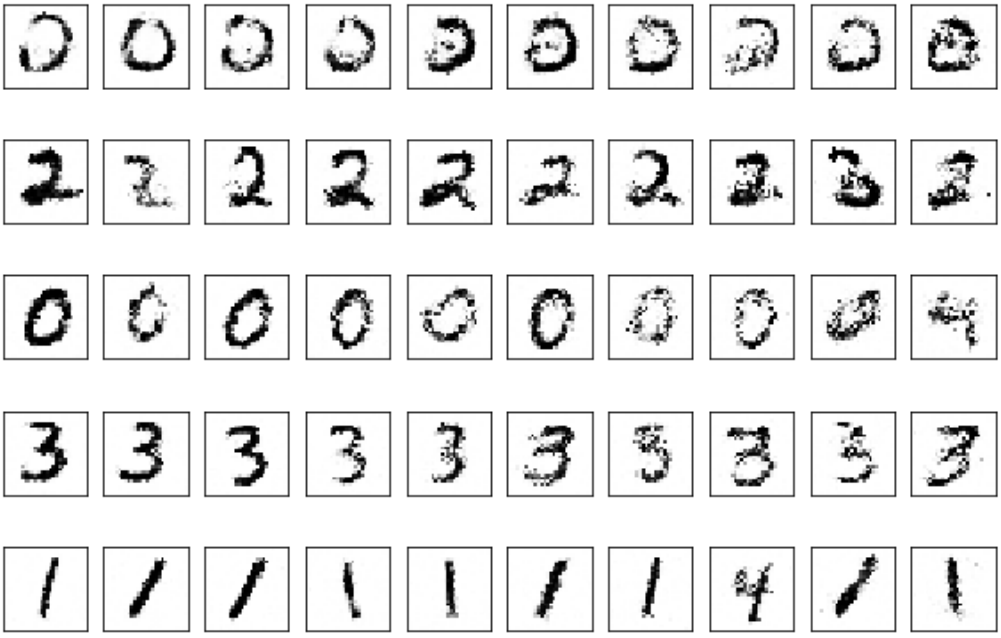


Figure 5.4: Samples from the fitted mixture components. Each row is sampled from the same component

cluster index true label	0	1	2	3	4
0	0.028702	0.002533	0.947662	0.019754	0.001351
1	0.000000	0.015426	0.001187	0.003263	0.980125
2	0.019470	0.854481	0.045821	0.019637	0.060591
3	0.001631	0.114011	0.046648	0.820910	0.016800
4	0.000685	0.005649	0.648237	0.000856	0.344574

Table 5.1: Normalized contingency table for the clustering induced by the model

From table 5.1 and figure 5.4 it's possible to see that although there is some confusion, the model successfully clusters the MNIST digits.

Chapter 6

Conclusions

6.1 Conclusions

Deep generative modelling is an active research avenue, and will keep being developed and improved, since it lends itself to extremely useful applications, like anomaly detection, synthetic data generation, and, generally speaking, uncovering patterns in data. Overall, the initial idea of the present work stands validated by the experiments: it is possible to learn mixtures of normalizing flows via the proposed procedure. The proposed method was tested on two synthetic datasets, succeeding with ease on one of them, and struggling with the other one. However, when allowed to learn from just a few labels, it was able to successfully fit to the data it previously failed on. On the real world dataset, the model's clustering capability was tested, as well as its ability to generate realistic samples, with some success. During the experiments, it became evident that, similarly to what happens with the majority of neural network based models, in order to successfully fit the proposed model to complex data some fine tuning is required, both in terms of the training procedure, as well as in terms of the architecture of the blocks that constitute the model. In the following section, some proposals and ideas for future work and for tackling some of the observed shortcomings are proposed.

6.2 Discussion and Future Work

After the work presented here, some observations and future research questions and ideas arise:

- The main shortcoming of the proposed model, specially in its fullt unsupervised variant, is that there is no way to incentivize the variational posterior to partition the space in the intuitively correct manner. Moreover, the variational posterior generally performs poorly in regions of space where there are few or no training points. This suggests that the model could benefit from a consistency loss regularization term. In fact, this idea has been pursued in [Izmailov et al., 2019].
- A weight-sharing strategy between components is also an interesting point for future research. It is plausible that, this way, components could share "concepts" and latent representations of data, and use their non-shared weights to "specialize" in their particular cluster of data. Take, for instance,

the Pinwheel dataset: in principle, the five normalizing flows could share a stack of layers that learned to model the concept of wing, each component then having a non-shared stack of blocks that would only need to model the correct rotation of its respective wing.

- During the experimentation phase it was found that a balance, between the complexity of the variational posterior and that of the components of the mixture, is crucial for the convergence to interesting solutions. This is intuitive: if the components are too complex, the variational posterior tends to ignore most of them and assigns most points to a single or few components. This balance of complexities is well studied in the case of GANs, for instance.
- The fact that in some cases the variational posterior ignores components and "chooses" not to use them can hypothetically be exploited in the scenarios where the number of clusters is unknown. If the dynamics of what drives the variational posterior to ignore components can be understood, perhaps they can be actively tweaked (via architectural choices, training procedure and hyperparameters, for example) to benefit the modelling task in such a scenario.
- The effect of using different architectures for the neural networks used was not evaluated. It is likely, for instance, that convolutional architectures would produce better results in the real world dataset.

Bibliography

Belkin, Mikhail et al. (Dec. 2018). “Reconciling modern machine learning and the bias-variance trade-off”. In: *arXiv*.

Chaitin, Gregory (Feb. 2016). “Doing Mathematics Differently”. In: *Inference - International Review of Science* Volume Two.Issue One.

De Cao, Nicola, Ivan Titov, and Wilker Aziz (2019). “Block Neural Autoregressive Flow”. In: *35th Conference on Uncertainty in Artificial Intelligence (UAI19)*.

Dilokthanakul, Nat et al. (2016). *Deep Unsupervised Clustering with Gaussian Mixture Variational Autoencoders*.

Dinh, Laurent, Jascha Sohl-Dickstein, and Samy Bengio (2017). “Density estimation using Real NVP”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Dinh, Laurent, Jascha Sohl-Dickstein, et al. (2019). *A RAD approach to deep mixture models*.

Germain, Mathieu et al. (July 2015). “MADE: Masked Autoencoder for Distribution Estimation”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, pp. 881–889.

Goodfellow, Ian et al. (2014). “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani et al. Curran Associates, Inc., pp. 2672–2680.

Grover, Aditya, Manik Dhar, and Stefano Ermon (2018). “Flow-GAN: Combining Maximum Likelihood and Adversarial Learning in Generative Models”. In: *AAAI Conference on Artificial Intelligence*.

Izmailov, Pavel et al. (2019). “Semi-Supervised Learning with Normalizing Flows”. In: *International Conference on Machine Learning. Workshop on Invertible Neural Networks and Normalizing Flows*.

Johnson, M. et al. (2016). “Composing graphical models with neural networks for structured representations and fast inference”. In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., pp. 2946–2954.

Kingma, Diederick P and Jimmy Ba (2015). “Adam: A method for stochastic optimization”. In: *International Conference on Learning Representations (ICLR)*.

Kingma, Diederik P and Max Welling (2014). “Auto-encoding variational Bayes”. In: *International Conference on Learning Representations (ICLR)*.

Kingma, Durk P and Prafulla Dhariwal (2018). “Glow: Generative Flow with Invertible 1x1 Convolutions”. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., pp. 10215–10224.

Lanterman, Aaron D. (Aug. 2001). “Schwarz, Wallace, and Rissanen: Intertwining Themes in Theories of Model Selection”. In: *International Statistical Review* 69.2, pp. 185–212. DOI: 10.1111/j.1751-5823.2001.tb00456.x.

LeCun, Yann and Corinna Cortes (2010). “MNIST handwritten digit database”. In:

Lin, Wu, Mohammad Emtiyaz Khan, and Nicolas Hubacher (2018). “Variational Message Passing with Structured Inference Networks”. In: *International Conference on Learning Representations*.

Papamakarios, George, Theo Pavlakou, and Iain Murray (2017). “Masked Autoregressive Flow for Density Estimation”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., pp. 2338–2347.

Paszke, Adam et al. (2017). “Automatic Differentiation in PyTorch”. In: *NIPS Autodiff Workshop*.

Rezende, Danilo and Shakir Mohamed (July 2015). “Variational Inference with Normalizing Flows”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, pp. 1530–1538.

Xie, Junyuan, Ross Girshick, and Ali Farhadi (20–22 Jun 2016). “Unsupervised Deep Embedding for Clustering Analysis”. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, pp. 478–487.

Zhang, Dejiao et al. (2017). *Deep Unsupervised Clustering Using Mixture of Autoencoders*.