Variational Mixture of Normalizing Flows

Guilherme Paulo Grijó Pires
Instituto Superior Técnico
mail@gpir.es
Lisboa, Portugal

***Abstract*—In the past few years, deep generative models, such as generative adversarial networks [1], variational autoencoders [2], and their variants, have seen wide adoption for the task of modelling complex data distributions. In spite of the outstanding sample quality achieved by those methods, they model the target distributions *implicitly*, in the sense that the probability density functions induced by them are not explicitly accessible. This fact renders those methods 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 - [3], [4]) a way to introduce discrete structure (such as the one found in mixtures) in the models it allows to construct, in an unsupervised scenario. The present work overcomes this 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 on a real-world dataset.**

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

I. Introduction

A. 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 [2] 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 distribution at 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 real 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 finite 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 to 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, Duvenaud, Wiltschko, *et al.* [6] and Lin, Khan, and Hubacher [7] 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, Mediano, Garnelo, *et al.* [8] proposes a VAE-inspired model, where the prior is a Gaussian mixture. Xie, Girshick, and Farhadi [9] describe 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 two previous publications that are most related to the present work are the following:

Dinh, Sohl-Dickstein, Pascanu, *et al.* [4], similarly to this work, try to reconcile normalizing flows with a multimodal/discrete structure. They do 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.

[1] In fact, recent work [5] 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.

Izmailov, Kirichenko, Finzi, *et al.* [3] also exploit 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), the method proposed by Izmailov, Kirichenko, Finzi, *et al.* [3] resort to a latent mixture of Gaussians as the base distribution for its flow model, and learn a single flow.

B. Objectives

The objectives of the present work can be summarized as follows:

- 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.

We have achieved these goals by proposing a method to learn a mixture of K normalizing flows, through the optimization of a variational inference objective, where the variational posterior is parameterized by a neural network with a softmax output, and its parameters are optimized jointly with those of the mixture components.

II. Notation

The main notation used throughout this work is as follows:

- Scalars and vectors are lower-case letters, with vectors in bold. E.g.: x is a scalar, $\mathbf{z}$ is a vector.
- Upper-case letters represent matrices.
- Vector $\boldsymbol{x}_{a:b}$ denotes the a-th to the b-th elements of vector $\boldsymbol{x}$.
- For distributions, subscript notation will only be used when the distribution is not 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.

III. Normalizing Flows

A. Introduction

The best-known and most 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, they allow for tractable parameter estimation, they have closed-form likelihood functions, and sampling from them is simple.

One way to obtain more expressive models is to assume the existence of latent variables, leverage certain factorization structures, and use well-known distributions for the individual factors of the product that constitutes the model's joint distribution. By using these structures and specific, well-chosen combinations of distributions (namely, conjugate prior-likelihood pairs), these models are able to remain 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 [10], and which has since evolved and developed into the basis of multiple state-of-the-art techniques for density modelling and estimation [11], [12], [13], [14].

B. Change of Variables

Given a random variable $\boldsymbol{z} \in \mathbb{R}^D$, with probability density function f_Z, and a bijective and continuous function $g(\,.\,;\boldsymbol{\theta}) : \mathbb{R}^D \to \mathbb{R}^D$, the probability density function f_X of the random variable $\boldsymbol{x} = g(\boldsymbol{z})$ is given by

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

$$= f_Z(g^{-1}(\boldsymbol{x};\boldsymbol{\theta}))\left|\det\left(\frac{d}{d\boldsymbol{z}}g(\boldsymbol{z};\boldsymbol{\theta})\Big|_{\boldsymbol{z}=g^{-1}(\boldsymbol{x};\boldsymbol{\theta})}\right)\right|^{-1}, \tag{2}$$

where $\det\left(\frac{d}{d\boldsymbol{x}}g^{-1}(\boldsymbol{x};\boldsymbol{\theta})\right)$ is the determinant of the Jacobian matrix of $g^{-1}(\,.\,;\boldsymbol{\theta})$, computed at $\boldsymbol{x}$. Since $g(\,.\,;\boldsymbol{\theta})$ 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 probability density function (also called *base density*). It is assumed that it has a closed-form expression. 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 both "cheap" to compute.

C. Normalizing Flows

Consider L transformations h_ℓ, for $\ell = 0, 1, ..., L-1$ that fulfill the three requirements listed above. Let each of those transformations be parameterizable by a parameter vector $\boldsymbol{\theta}_\ell$, for $\ell = 0, 1, ..., L-1$. The dependence on the parameter vectors will be implicit from here on. 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. Notice that, with this notation, $\boldsymbol{z_L} = \boldsymbol{x}$. Furthermore, let g

be the composition of the L transformations. Applying the change of variables formula to

$$\boldsymbol{z_0} \sim f_Z \tag{3}$$
$$\boldsymbol{x} = h_{L-1} \circ h_{L-2} \circ ... \circ h_0(\boldsymbol{z_0}), \tag{4}$$

noting that $g^{-1} = h_0^{-1} \circ h_1^{-1} \circ ... \circ h_{L-1}^{-1}$ and using the chain rule for derivatives, leads to

$$f_X(\boldsymbol{x}) = f_Z(g^{-1}(\boldsymbol{x}))\Big|\det\Big(\frac{d}{d\boldsymbol{x}}g^{-1}(\boldsymbol{x})\Big)\Big| \tag{5}$$
$$= f_Z(g^{-1}(\boldsymbol{x}))\prod_{\ell=0}^{L-1}\Big|\det\Big(\frac{d}{d\boldsymbol{z_{\ell+1}}}h_\ell^{-1}(\boldsymbol{z_{\ell+1}})\Big)\Big| \tag{6}$$
$$= f_Z(g^{-1}(\boldsymbol{x}))\prod_{\ell=0}^{L-1}\Big|\det\Big(\frac{d}{d\boldsymbol{x_\ell}}h_\ell(\boldsymbol{x_\ell})\Big|_{\boldsymbol{x_\ell}=h_\ell^{-1}(\boldsymbol{z_{\ell+1}})}\Big)\Big|^{-1} \tag{7}$$

Replacing $h_\ell^{-1}(\boldsymbol{z_{\ell+1}}) = \boldsymbol{z_\ell}$ in (7) leads to

$$f_X(\boldsymbol{x}) = f_Z(g^{-1}(\boldsymbol{x}))\prod_{\ell=0}^{L-1}\Big|\det\Big(\frac{d}{d\boldsymbol{z_\ell}}h_\ell(\boldsymbol{z_\ell})\Big)\Big|^{-1}; \tag{8}$$

taking the logarithm,

$$\log f_X(\boldsymbol{x}) = \log f_Z(g^{-1}(\boldsymbol{x})) - \sum_{\ell=0}^{L-1}\log\Big|\det\Big(\frac{d}{d\boldsymbol{z_\ell}}h_\ell(\boldsymbol{z_\ell})\Big)\Big|. \tag{9}$$

Depending on the task, one might prefer to replace the second term in (9) with a sum of log-absolute-determinants of the Jacobians of the inverse transformations. This choice would imply replacing the minus sign before the sum with a plus sign:

$$\log f_X(\boldsymbol{x}) =$$
$$= \log f_Z(g^{-1}(\boldsymbol{x})) + \sum_{\ell=0}^{L-1}\log\Big|\det\Big(\frac{d}{d\boldsymbol{z_{\ell+1}}}h_\ell^{-1}(\boldsymbol{z_{\ell+1}})\Big)\Big|. \tag{10}$$

We started by assuming that the transformations h_ℓ fulfill the requirements listed in Section III-B. For that reason, it is clear that the above expression is a feasible objective for gradient-based optimization. In practice, this is carried out by leveraging modern automatic differentiation and optimization frameworks [11], [12], [15]. 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 can be used as flexible variational posteriors, in variational inference settings, as well as density estimators.

D. Examples of transformations

1) Affine Transformation: An affine transformation is arguably the simplest choice; it 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 $\boldsymbol{b}$:

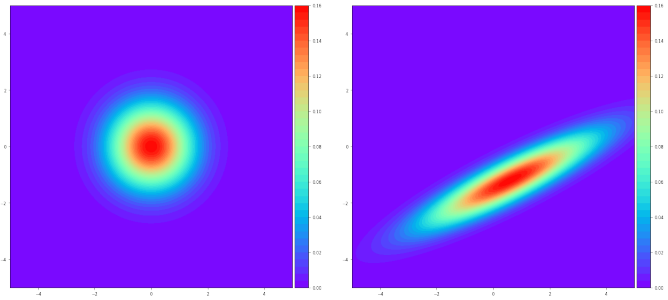
$$\boldsymbol{z} \sim p(\boldsymbol{z}) \tag{11}$$
$$\boldsymbol{x} = A\boldsymbol{z} + \boldsymbol{b}. \tag{12}$$

The determinant of the Jacobian of this transformation is simply the determinant of A. However, in general, computing the determinant of a $D \times D$ matrix has $\mathcal{O}(D^3)$ computational complexity. For that reason, it is common to use matrices with a certain structure that makes their determinants easier to compute. For instance, if A is triangular, its determinant is 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 $\mathcal{O}(D)$ complexity and that are more expressive than simple triangular matrices. Kingma and Dhariwal [11] propose one such transformation. It constrains the matrix A to be decomposable as $A = PL\big(U + \text{diag}(\boldsymbol{s})\big)$, where $\text{diag}(\boldsymbol{s})$ is a diagonal matrix whose diagonal's elements are the components of vector $\boldsymbol{s}$. The following additional constrains are in place:

- 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 matrix A is simply the product of the elements of $\boldsymbol{s}$.



(a) (b)

Figure 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)

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

$$f(\boldsymbol{z}) = [f_1(z_1), f_2(z_2), ..., f_D(z_D)], \tag{13}$$

where

$$f_i(z_i) = \begin{cases} z_i, & \text{if } z_i \geq 0, \\ \alpha z_i, & \text{otherwise.} \end{cases} \tag{14}$$

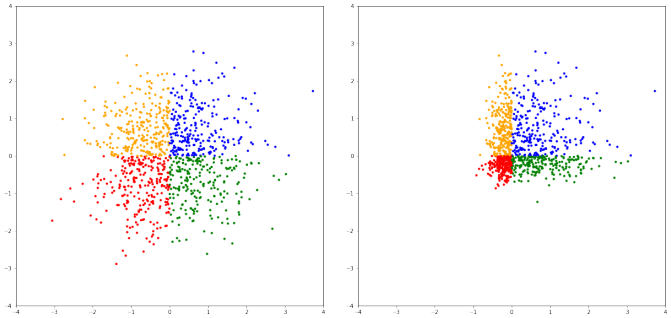
In order for the transformation to be invertible, it is necessary that $\alpha > 0$. Let us define a function $j(.)$ as

$$j(z_i) = \begin{cases} 1, & \text{if } z_i \geq 0, \\ \alpha, & \text{otherwise;} \end{cases} \tag{15}$$

it is trivial to see that the Jacobian of the transformation is a diagonal matrix, whose diagonal elements are $j(z_i)$:

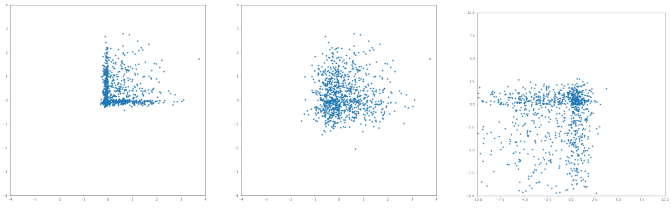
$$J(f(z)) = \begin{bmatrix} j(z_1) & & & \\ & j(z_2) & & \\ & & \ddots & \\ & & & j(z_D) \end{bmatrix}. \tag{16}$$

With that in hand, it is easy to arrive at the log-absolute-determinant of this transformation's Jacobian, which is given by $\sum_{i=1}^{D} \log \left| j(z_i) \right|$



(a) (b)

Figure 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: 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) Batch-Normalization Transformation: Dinh, Sohl-Dickstein, and Bengio [12] 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 $\tilde{\sigma}^2$:

$$f(z) = \frac{z - \tilde{\mu}}{\sqrt{\tilde{\sigma}^2 + \epsilon}}, \tag{17}$$

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

$$\prod_{i=1}^{D} \frac{1}{\sqrt{\tilde{\sigma}_i^2 + \epsilon}}. \tag{18}$$

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 sufficiently expressive and whose Jacobians are not computationally heavy. One brilliant example of such transformations, proposed by Dinh, Sohl-Dickstein, and Bengio [12], 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 $t(.)$. The transformation is defined as:

$$\begin{cases} \boldsymbol{x_1} &= \boldsymbol{z_1} \odot \exp\big(s(\boldsymbol{z_2})\big) + t(\boldsymbol{z_2}) \\ \boldsymbol{x_2} &= \boldsymbol{z_2}. \end{cases} \tag{19}$$

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

- $\frac{\partial \boldsymbol{x_2}}{\partial \boldsymbol{z_2}} = I$, 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, as will become clear ahead.

Writing the above in matrix form:

$$J_{f(z)} = \begin{bmatrix} \frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_1}} & \frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_2}} \\ \frac{\partial \boldsymbol{x_2}}{\partial \boldsymbol{z_1}} & \frac{\partial \boldsymbol{x_2}}{\partial \boldsymbol{z_2}} \end{bmatrix} \tag{20}$$

$$= \begin{bmatrix} \text{diag}\Big(\exp\big(s(\boldsymbol{z_2})\big)\Big) & \frac{\partial \boldsymbol{x_1}}{\partial \boldsymbol{z_2}} \\ \boldsymbol{0} & I \end{bmatrix} \tag{21}$$

shows that the Jacobian matrix is (upper) 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

$$\det\left(J_{f(z)}\right) = \prod_i \exp\left(s(\boldsymbol{z_2^{(i)}})\right) \tag{22}$$

$$\log\left|\det\left(J_{f(z)}\right)\right| = \sum_i s(\boldsymbol{z_2^{(i)}}), \tag{23}$$

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.

5) Masked Autoregressive Flows: Another ingenious architecture for normalizing flows has been proposed by Papamakarios, Pavlakou, and Murray [14]. 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:

$$x_i = z_i \exp(\alpha_i) + \mu_i \tag{24}$$

$$(\mu_i, \alpha_i) = g(\boldsymbol{x_{1:i-1}}). \tag{25}$$

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, μ, and a log-scale, α, which translates to the fact that the function g itself does not need to be inverted:

$$z_i = (x_i - \mu_i)\exp(-\alpha_i). \tag{26}$$

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

$$\det\left(J_{f(\boldsymbol{z})}\right) = \prod_{i=1}^{D} \exp(\alpha_i), \tag{27}$$

$$\log\left|\det\left(J_{f(\boldsymbol{z})}\right)\right| = \sum_{i=1}^{D} \alpha_i. \tag{28}$$

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 a masked autoencoder for distribution estimation (MADE), as described by Germain, Gregor, Murray, *et al.* [16].

Much like the partitioning in the affine coupling layer, the assumption of autoregressiveness (and the ordering of the elements of $\boldsymbol{x}$ 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.

E. 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. 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. In general, 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[2]. To fit a normalizing flow via MLE, a gradient based optimizer is used to minimize $\hat{\mathcal{L}}(\boldsymbol{\theta}) = -\mathbb{E}[\log p(\boldsymbol{x}|\boldsymbol{\theta})]$. However, this expectation is generally not accesible, since we have only finite samples of $\boldsymbol{x}$. Because of that, the parameters are estimated by optimizing an approximation of that expectation: $-\frac{1}{N}\sum_{i=1}^{N} \log p(\boldsymbol{x_i}|\boldsymbol{\theta})$.

To perform optimization on this objective, stochastic gradient descent (SGD) - and its variants - is the most commonly used algorithm. In general terms, SGD is an approximation of gradient descent, which rather than using the actual gradient, at time step t, to update the variables under optimization, works by computing several estimates of that gradient and using those estimates instead. This is done by partitioning the data in mini-batches, and computing the loss function and respective gradients over those mini-batches. This way, one pass through data - an *epoch* - results in several parameter updates.

IV. Variational Mixture of Normalizing Flows

A. Introduction

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.

B. Model Definition

Let us define a mixture model, where each of the K components is a density parameterized by a normalizing flow. For simplicity, consider that all of the K normalizing flows have the same architecture[3], 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 K-class softmax output, with parameters $\boldsymbol{\gamma}$. This network will

[2]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.

[3]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 the most flexible normalizing flow is "allocated" to the most complex cluster. This is an interesting direction for future research.

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 (the dependence of q on x is made explicit):

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

Let us rearrange it:

$$\begin{aligned}\text{ELBO} &= \mathbb{E}_q[\log p(\boldsymbol{x}|z)] + \mathbb{E}_q[\log p(z)] - \mathbb{E}_q[\log q(z|\boldsymbol{x})] \quad (29)\\ &= \mathbb{E}_q[\log p(\boldsymbol{x}|z) + \log p(z) - \log q(z|\boldsymbol{x})] \quad (30)\end{aligned}$$

Since $q(z|\boldsymbol{x})$ is given by the forward-pass of a neural network, and is therefore straightforward to obtain, the expectation in (30) 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:

$$\text{ELBO} = \sum_{z=1}^{K} q(z|\boldsymbol{x})\big(\log p(\boldsymbol{x}|z) + \log p(z) - \log q(z|\boldsymbol{x})\big). \quad (31)$$

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 section 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.
- $-\log q(z|\boldsymbol{x})$ is the negative logarithm of the output of the encoder.

Let us call this model *variational mixture of normalizing flows* (VMoNF). For an overview of the model, consider Figures 4 and 5

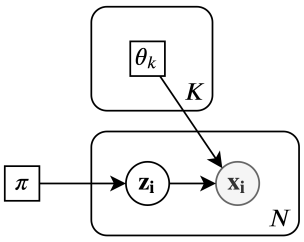


Figure 4: Plate diagram of a mixture of K normalizing flows. $\boldsymbol{\theta}_k$ is the parameter vector of component k.

In a similar fashion to the variational auto-encoder, proposed by Kingma and Welling [2], 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 naturally induces a clustering on the data, and can be directly used to assign new data points to the discovered clusters. Moreover, each of the fitted components can be used to generate samples from the cluster it "specialized" in.

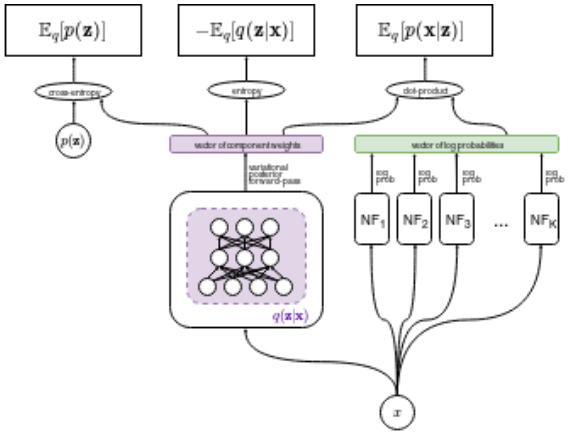


Figure 5: Overview of the training procedure of the VMoNF.

C. Implementation

To implement and test the proposed model, Python was the chosen language. More specifically, this work heavily relies on the PyTorch [17] package and framework for automatic differentiation. Moreover, the parameter optimization is done via stochastic optimization, namely using the Adam optimizer, proposed by Kingma and Ba [18].

Figure 5 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.

V. Experiments

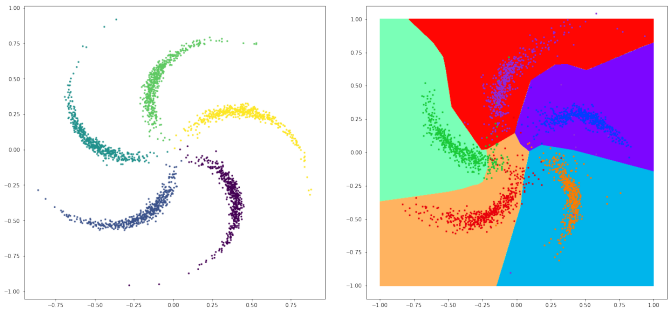
In this section, the proposed model is applied to two benchmark synthetic datasets (Pinwheel and Two-circles) and one real-world dataset (MNIST). On one of the synthetic datasets, one shortcoming of the model is brought to attention, but is overcome in a semi-supervised setting. On the real-world dataset,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

A technique inspired in the work of Zhang, Sun, Eriksson, *et al.* [19] was employed to improve training speed and quality of results. This consists in dividing the inputs of the softmax layer in the variational posterior by a "temperature" value, T, which follows an exponential decay schedule during training. Intuitively, this makes the variational posterior "more certain" as training proceeds, while allowing all components to be generally exposed to the whole data, during the initial epochs. This discourages components from being "subtrained" during

the initial epochs and, subsequently, from being prematurely discarded by the variational posterior.

A. Toy datasets

1) Pinwheel dataset: This dataset is constituted by five non-linear "wings". See Figure 6 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.



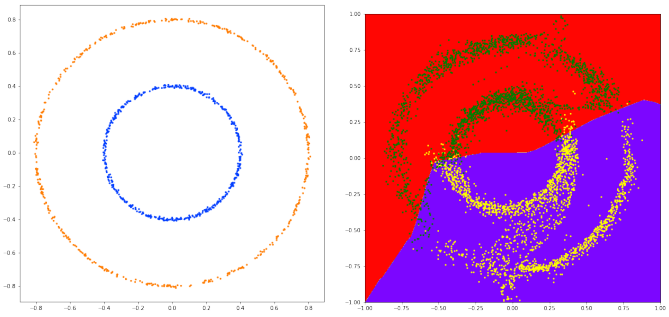
(a) (b)

Figure 6: (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 the dots would not be visible)

This experiment consisted of training on 2560 data points (512 per class) using the Adam optimizer, with a learning rate of 0.001, 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 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.

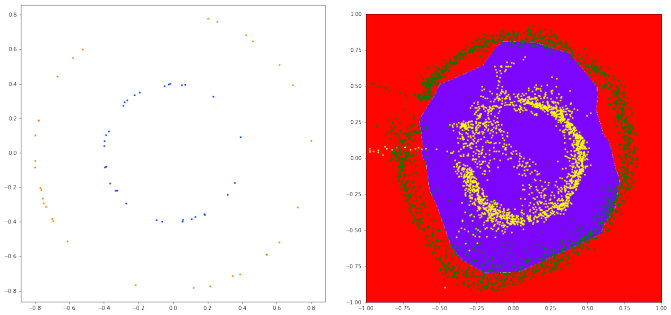
2) Two-circles dataset: This dataset consists of two concentric circles. The experiment on this dataset, shown on Figure 7, makes evident one shortcoming of this model: the way in which the variational posterior partitions the space is not necessarily guided by the intrisic structure in the data. In the case of the two-circles dataset, it was found that the most common space partitioning induced by the model consisted simply of splitting into two half-spaces. However, in a semi-supervised setting, this behaviour can be corrected and the model successfully learns to separate the two circles, as shown in Figure 8. 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 8, 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. Moreover, the points in diametrically opposed regions of the same circle are more dissimilar (in the geometrical sense) than points in the same region of the two circles. Therefore, when completely uninformed by labels, the variational posterior's layers will tend to have similar activations for points in the latter case, and thus tend to place them in the same class.

The unsupervised learning experiment consisted of training on 1024 datapoints, 512 per class; using the Adam optimizer, with a learning rate of 0.001, 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 64 labeled data points, 32 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 optimizer, with a learning rate of 0.001, 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 7: (a) Original dataset. (b) Samples from the learned model, without any labels. Coloring logic is the same as in 6.



(a) (b)

Figure 8: (a) Labeled points used in semi-supervised scenario. (b) Samples from the model trained in the semi-supervised scenario.

B. Real-world dataset

In this subsection, the proposed model is evaluated on the well-known MNIST dataset [20]. This dataset consists 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 9, 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.

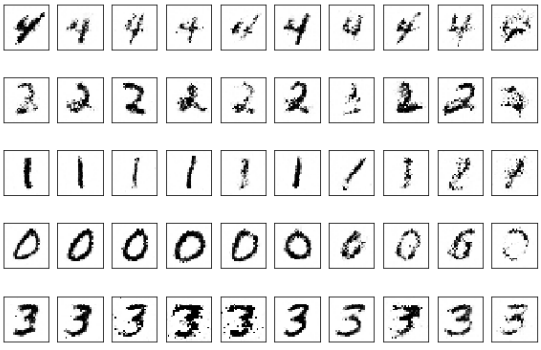


Figure 9: Samples from the fitted mixture components. Each row is sampled from the same component

From Table I and Figure 9 it is possible to see that although there is some confusion, the model successfully clusters the MNIST digits.

True label \ Cluster index	0	1	2	3	4
0	0.000602	0.012432	0.002807	0.982555	0.001604
1	0.002139	0.020146	0.977001	0.000178	0.000535
2	0.000802	0.952276	0.011630	0.007219	0.028073
3	0.001558	0.479455	0.300682	0.004284	0.214021
4	0.646166	0.347273	0.005125	0.001435	0.000000

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

VI. Conclusions

A. Conclusions

Deep generative modelling is an active research avenue that 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 - as well as by recently published similar work [4, 3]. 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 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 subsection, some proposals and ideas for future work and for tackling some of the observed shortcomings are proposed.

B. 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 fully 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 by Izmailov, Kirichenko, Finzi, *et al.* [3].
- Some form of 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.
- 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.
- Related to the previous point, one first experiment could be to update the prior ($p(z)$) (for example, every epoch), based on the responsabilities given by the variational posterior.
- 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.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets", in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2014, pp. 2672–2680.

[2] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes", in *International Conference on Learning Representations (ICLR)*, 2014.

[3] P. Izmailov, P. Kirichenko, M. Finzi, and A. G. Wilson, "Semi-Supervised Learning with Normalizing Flows", in *International Conference on Machine Learning, Workshop on Invertible Neural Networks and Normalizing Flows*, 2019.

[4] L. Dinh, J. Sohl-Dickstein, R. Pascanu, and H. Larochelle, *A rad approach to deep mixture models*, 2019.

[5] A. Grover, M. Dhar, and S. Ermon, "Flow-gan: Combining maximum likelihood and adversarial learning in generative models", in *AAAI Conference on Artificial Intelligence*, 2018.

[6] M. Johnson, D. K. Duvenaud, A. Wiltschko, R. P. Adams, and S. R. Datta, "Composing graphical models with neural networks for structured representations and fast inference", in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., Curran Associates, Inc., 2016, pp. 2946–2954.

[7] W. Lin, M. E. Khan, and N. Hubacher, "Variational message passing with structured inference networks", in *International Conference on Learning Representations*, 2018.

[8] N. Dilokthanakul, P. A. M. Mediano, M. Garnelo, M. C. H. Lee, H. Salimbeni, K. Arulkumaran, and M. Shanahan, *Deep unsupervised clustering with gaussian mixture variational autoencoders*, 2016.

[9] J. Xie, R. Girshick, and A. Farhadi, "Unsupervised deep embedding for clustering analysis", in *Proceedings of The 33rd International Conference on Machine Learning*, M. F. Balcan and K. Q. Weinberger, Eds., ser. Proceedings of Machine Learning Research, vol. 48, New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 478–487.

[10] D. Rezende and S. Mohamed, "Variational inference with normalizing flows", in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, Jul. 2015, pp. 1530–1538.

[11] D. P. Kingma and P. Dhariwal, "Glow: Generative flow with invertible 1x1 convolutions", in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., Curran Associates, Inc., 2018, pp. 10 215–10 224.

[12] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real NVP", in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[13] N. De Cao, I. Titov, and W. Aziz, "Block neural autoregressive flow", *35th Conference on Uncertainty in Artificial Intelligence (UAI19)*, 2019.

[14] G. Papamakarios, T. Pavlakou, and I. Murray, "Masked autoregressive flow for density estimation", in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 2338–2347.

[15] J. Ho, X. Chen, A. Srinivas, Y. Duan, and P. Abbeel, "Flow++: Improving flow-based generative models with variational dequantization and architecture design", in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, Long Beach, California, USA: PMLR, Sep. 2019, pp. 2722–2730.

[16] M. Germain, K. Gregor, I. Murray, and H. Larochelle, "Made: Masked autoencoder for distribution estimation", in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, Jul. 2015, pp. 881–889.

[17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch", in *NIPS Autodiff Workshop*, 2017.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", in *International Conference on Learning Representations (ICLR)*, 2015.

[19] D. Zhang, Y. Sun, B. Eriksson, and L. Balzano, *Deep unsupervised clustering using mixture of autoencoders*, 2017.

[20] Y. LeCun and C. Cortes, "MNIST handwritten digit database", 2010.