# Normalizing Flows

Abdul Fatir Ansari, Devamanyu Hazarika, and Remmy A. M. Zen

National University of Singapore
{abdulfatir,devamanyu,remmy}@u.nus.edu

## 1 Introduction

Improving inference, sampling, and density estimation in deep generative models (DGMs) are important research problems. Simple distributions (e.g., Gaussian) are often used as likelihood distributions in DGMs. However, the true distribution is often far from this simple distribution, which results in issues such as blurry reconstructions in the case of images. Latent variable models, such as VAEs, often set the variational posterior distribution $q(\mathbf{z}|\mathbf{x})$ to a factorial multivariate Gaussian distribution. Such a simplistic assumption hampers the model in multiple ways. For instance, this does not allow a multi-modal latent space distribution. Normalizing Flows allow transformation of samples from a simple distribution into samples from a complex distribution, whose density can be evaluated analytically, by applying a series of invertible transformations. In this report, we discuss a number of recent works that introduce techniques for improved variational inference and/or density estimation in deep generative models using normalizing flows.

## 2 Background

### 2.1 Autoencoder

An Autoencoder (AE) [7] is comprised of two neural networks: 1) The encoder network $f_\phi$ which transforms an input data-point $\mathbf{x} \in \mathbb{R}^d$ into a representation $\mathbf{z} \in \mathbb{R}^p$ where $p < d$ generally; 2) The decoder network $g_\theta$ which attempts to reconstruct $\mathbf{x}$ as $\hat{\mathbf{x}} \in \mathbb{R}^d$ from the representation $\mathbf{z}$. Training proceeds by minimizing the reconstruction error between $\mathbf{x}$ and $\hat{\mathbf{x}}$ and the training objective is given in Eq. (1)

$$\phi, \theta = \arg\min_{\phi,\theta} \mathcal{L}(\hat{\mathbf{x}}, \mathbf{x}) \tag{1}$$

where $\mathcal{L}$ is a suitable reconstruction error which can be the squared error $\|\hat{\mathbf{x}} - \mathbf{x}\|^2$ for real-valued observations or binary cross-entropy $\sum_{i=1}^{d} -\mathbf{x}|_i \log(\hat{\mathbf{x}}|_i) - (1 - \mathbf{x}|_i) \log(1 - \hat{\mathbf{x}}|_i)$ for binary observations.

### 2.2 Variational Autoencoder

The Variational Autoencoder (VAE) [10] combines an inference network $f_\phi$ (analogous to the encoder in AE) with a generative model $g_\theta$ (analogous to the decoder in AE). VAE maximizes the *evidence lower bound* (ELBO) which can be derived as follows

$$\log p(\mathbf{x}) = \log \int_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \log \int_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}) \frac{q_\phi(\mathbf{z}|\mathbf{x})}{q_\phi(\mathbf{z}|\mathbf{x})} d\mathbf{z} \tag{2}$$

$$\geq \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} \left[ \log \frac{p_\theta(\mathbf{x}|\mathbf{z}) p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \right] = \mathcal{L}_{\text{ELBO}} \tag{3}$$

where the last inequality comes from Jensen's inequality.

A data-point $\mathbf{x} \in \mathbb{R}^d$ is passed through the inference network $f_\phi$ which outputs the parameters of the variational distribution $q_\phi(\mathbf{z}|\mathbf{x})$. A point $\mathbf{z} \in \mathbb{R}^p$ is sampled from $q_\phi(\mathbf{z}|\mathbf{x})$ and is passed onto the generative model $g_\theta$ which attempts to reconstruct $\mathbf{x}$ as $\hat{\mathbf{x}} \in \mathbb{R}^d$. Once training is complete, the inference network and the generative model can be used separately. New samples from the VAE can be generated by sampling $\mathbf{z}$ from the prior $p(\mathbf{z})$ (generally set to $\mathcal{N}(\mathbf{0}, \mathbf{I})$) and plugging into the generative model.

Maximization of $\mathcal{L}_{\text{ELBO}}$ using stochastic gradient descent (SGD) requires computation of $\nabla_\phi \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} \left[ \log \frac{p_\theta(\mathbf{x}|\mathbf{z}) p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})} \right]$ which is done using the *reparameterization trick*. Briefly, the reparameterization trick uses an invertible standardization function $\mathcal{S}_\phi(\mathbf{z})$ such that $\mathcal{S}_\phi(\mathbf{z}) = \epsilon$ where $\epsilon \sim p(\epsilon)$ and $p(\epsilon)$ is a simple distribution. For example, $\mathcal{S}_\phi(\mathbf{z})$ for a normally distributed random variable would be $\mathcal{S}_\phi(\mathbf{z}) = \frac{\mathbf{z}-\boldsymbol{\mu}}{\boldsymbol{\sigma}} = \epsilon$ with $p(\epsilon)$ being $\mathcal{N}(\mathbf{0}, \mathbf{I})$. Once such an invertible standardization function is available, $\nabla_\phi \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})} [f(\mathbf{z})]$ can be written as $\nabla_\phi \mathbb{E}_{p(\epsilon)} \left[ f(\mathcal{S}_\phi^{-1}(\epsilon)) \right]$ which can then be evaluated by using the chain rule as $\mathbb{E}_{p(\epsilon)} \left[ \nabla_{\mathbf{z}} f(\mathcal{S}_\phi^{-1}(\epsilon)) \nabla_\phi \mathcal{S}_\phi^{-1}(\epsilon) \right]$. For more details please refer to Figurnov et al. [5].

## 3 Normalizing Flows

Before defining normalizing flows, let's consider a univariate distribution with density function $p(x)$. Define a continuous, differentiable, and increasing function $f$. Define $y = f(x)$ where $x \sim p(x)$. The density function of the random variable $Y$ can then be derived analytically using the Cumulative Distribution Function (CDF) as follows.

$$F_Y(y) = P(Y \leq y) \tag{4}$$

$$= P(f(X) \leq y) \tag{5}$$

$$= P(X \leq f^{-1}(y)) = F_X(f^{-1}(y)) \tag{6}$$

We end up with the CDF of the random variable $X$ at the point $f^{-1}(y)$. Now, $p(y) = F'_Y(y)$ by definition, where

$$F_Y(y) = F_X(f^{-1}(y)) = \int_{-\infty}^{f^{-1}(y)} p(x)dx \tag{7}$$

Differentiating Eq. (7) with respect to $y$ (using the Fundamental Theorem of Calculus and the chain rule) we get

$$p(y) = p(f^{-1}(y)) \cdot \frac{df^{-1}}{dy} \tag{8}$$

When $f$ is a decreasing function, we get $p(y) = -p(f^{-1}(y)) \cdot \frac{df^{-1}}{dy}$. For an invertible function in general, Eq. (8) can be written as

$$p(y) = p(f^{-1}(y)) \cdot \left|\frac{df^{-1}}{dy}\right| \tag{9}$$

Eq. (9) can be extended to the multivariate case where the derivative is replaced by the determinant of the Jacobian matrix

$$p(\mathbf{y}) = p(f^{-1}(\mathbf{y})) \cdot \left|\det \frac{\partial f^{-1}}{\partial \mathbf{y}}\right| = p(f^{-1}(\mathbf{y})) \cdot \left|\det \frac{\partial f}{\partial f^{-1}(\mathbf{y})}\right|^{-1} \tag{10}$$

In the above equation, the second equality comes from the inverse function theorem. Successive applications of such smooth, invertible transformation on a random variable with known density is called a *normalizing flow*.

Computation of the probability density of the transformed random variable requires the computation of the determinant of the Jacobian matrix which is computationally expensive as it scales with $O(d^3)$ where $d$ is the dimensionality of the random variable. Developing transformations with cheap determinant computation has been the primary focus of many recent works.

## 4 Applications

Literature on normalizing flows can be broadly classified into two parts: ones using normalizing flows for improved variational inference and ones using normalizing flows for density estimation.

### 4.1 Variational Inference

Variational methods perform inference by approximating the true posterior $p(\mathbf{z}|\mathbf{x})$ using a simpler variational family $q_\phi(\mathbf{z}|\mathbf{x})$. Recent works have focused on improving the variational posterior used in the VAE which is generally set to a

multivariate normal distribution with diagonal covariance matrix $\mathcal{N}(\boldsymbol{\mu}, \sigma^2\mathbf{I})$. It is clear that such a simplistic, unimodal choice for the posterior can be arbitrarily far away from the true posterior which can be a complex multi-modal distribution.

Recent works seek to convert samples from a simple variational posterior (such as the multivariate normal distribution) into a richer distribution by applying a series of smooth, invertible transformations or a flow. Let $\mathbf{z}_0$ be a sample from a simple distribution $q_0(\mathbf{z}_0)$ and $\mathbf{z}_K$ be a sample obtained by applying a flow of length $K$ on $\mathbf{z}_0$, i.e., $\mathbf{z}_K = f_K \circ f_{K-1} \circ \cdots \circ f_1(\mathbf{z}_0)$. Using Eq. (10), the density function $q_K(\mathbf{z}_K)$ is given by

$$q_K(\mathbf{z}_K) = q_0(\mathbf{z}_0) \prod_{k=1}^{K} \left| \det \frac{\partial f_k}{\partial \mathbf{z}_{k-1}} \right|^{-1} \tag{11}$$

The variational lower bound in VAEs (Eq. 3) can now be modified by setting $q(\mathbf{z}|\mathbf{x}) = q_K(\mathbf{z}_K|\mathbf{x})$

$$\mathcal{L} = \mathbb{E}_{q(\mathbf{z}_K|\mathbf{x})} \left[ \log p_\theta(\mathbf{x}, \mathbf{z}_K) - \log q(\mathbf{z}_K|\mathbf{x}) \right] \tag{12}$$

$$= \mathbb{E}_{q(\mathbf{z}_0|\mathbf{x})} \left[ \log p_\theta(\mathbf{x}, \mathbf{z}_K) - \log q(\mathbf{z}_K|\mathbf{x}) \right] \tag{13}$$

where $q(\mathbf{z}_0|\mathbf{x})$ is the simple initial density. Plugging in Eq. (11) into Eq. (13), we get a modified bound for flow-based VAEs

$$\mathcal{L} = \mathbb{E}_{q_0(\mathbf{z_0}|\mathbf{x})} \left[ \log p(\mathbf{x}, \mathbf{z}_K) - \log q_0(\mathbf{z}_0|\mathbf{x}) + \sum_{k=1}^{K} \log \left| \det \frac{\partial f_k}{\partial \mathbf{z}_{k-1}} \right| \right] \tag{14}$$

**Planar and Radial Flow** Planar and Radial Flows [13] are one of the earliest flows proposed in the context of variational inference. Planar flows apply transformations perpendicular to a plane while radial flows apply them around a point.

Planar flows use functions of the form

$$f(\mathbf{z}) = \mathbf{z} + \mathbf{u}h(\mathbf{w}^\top \mathbf{z} + b) \tag{15}$$

where $\mathbf{u}, \mathbf{w} \in \mathbb{R}^d$, $b \in \mathbb{R}$, and $h$ is an element-wise non-linearity such as tanh. The Jacobian determinant is then given by

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = (1 + h'(\mathbf{w}^\top \mathbf{z} + b)\mathbf{w}^\top \mathbf{u}) \tag{16}$$

which can be computed in $O(d)$ time.

Radial flows use functions of the form

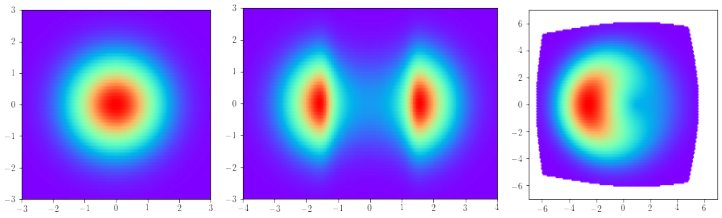
$$f(\mathbf{z}) = \mathbf{z} + \beta h(\alpha, r)(\mathbf{z} - \mathbf{z}_0) \tag{17}$$

where $\alpha \in \mathbb{R}^+$, $\beta \in \mathbb{R}$, $h(\alpha, r) = (\alpha + r)^{-1}$ and $r = ||\mathbf{z} - \mathbf{z}_0||$.

The Jacobian determinant is then given by

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = (1 + \beta h(\alpha, r) + \beta h'(\alpha, r) r) \left(1 + \beta h(\alpha, r)\right)^{d-1} \tag{18}$$

For a detailed derivation of Jacobians of Planar and Radial flows please refer the appendix. Fig. 1 shows how planar and radial flows change a standard normal density.



(a) Initial Density (b) Planar Flow (c) Radial Flow

Fig. 1: Change in standard normal density on application of length 1 planar and radial flows.

**Inverse Autoregressive Flow** Planar and radial flows provide a simple invertible transformation shown to be effective in a low-dimensional latent spaces (up to hundred dimensions). The transformation in planar flows (Eq. 15) can be seen as a Multilayer Perceptron (MLP) with a single-unit bottleneck hidden layer with a skip connection. Since a single-unit hidden layer isn't very expressive, a long chain of transformations is needed to model a high-dimensional distribution.

Autoregressive flows [6] is a normalizing flow that scales to high-dimensional latent space by exploiting the ordering f the variables. In autoregressive flow, given a sequence of variable $\mathbf{y} = y_{i}{}_{i=0}^{D}$, each variable is only dependent only on variables from the previous index. The distribution is then given by

$$p(\mathbf{y}) = \prod_{i=0}^{D} p(x_i | x_0, \ldots, x_{i-1}) \tag{19}$$

Kingma et. al. [9] proposed a Gaussian version of an autoregressive flow on a noise vector $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ given as follow:

$$y_0 = \mu_0 + \sigma_0 \epsilon_0 \tag{20}$$

$$y_i = \mu_i(\mathbf{y}_{0:i-1}) + \sigma(\mathbf{y}_{0:i-1}) \epsilon_i \tag{21}$$

This flow is invertible and the noise $\epsilon$ is given by:

$$\epsilon_i = \frac{y_i - \mu_i(\mathbf{y}_{0:i-1})}{\sigma(\mathbf{y}_{0:i-1})} \tag{22}$$

Note that, *epsilon* is independent to each other so the calculation of Equation 22 can be vectorized as follow:

$$\boldsymbol{\epsilon} = \frac{\mathbf{y} - \boldsymbol{\mu}(\mathbf{y})}{\boldsymbol{\sigma}(\mathbf{y})} \tag{23}$$

This enables an efficient computation with GPU.

Due to the autoregressive structure, the transformation has a lower triangular Jacobian where diagonal is $\sigma_i$. For calculation of normalizing flows, we are interested in log-determinant of the Jacobian which is just a product of the diagonal given as:

$$\log\det\left|\frac{d\boldsymbol{\epsilon}}{d\mathbf{y}}\right| = \sum_{i=0}^{D} -\log\sigma_i(\mathbf{y}) \tag{24}$$

To apply Inverse Autoregressive Flows (IAF) for variational inference in VAE, we add IAF transforms after the latent variables $\mathbf{z}$ and modify the likelihood to account for IAF transforms. Figure 2 shows the process of applying IAF to Variational Autoencoder.

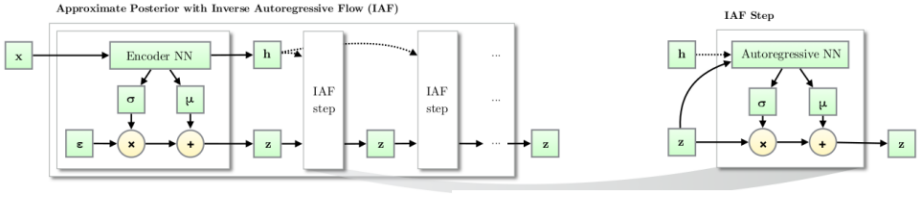



Fig. 2: The process of Inverse Autoregressive Flows in Variational Autoencoder [9].

The flow consists of a chain of $T$ following transformations:

$$\mathbf{z}_t = \boldsymbol{\mu}_t + \boldsymbol{\sigma}_t \mathbf{z}_{t-1} \tag{25}$$

Kingma et.al. [9] proposed a more stable update based on LSTM-type update. LSTM is a type of recurrent neural network that applies autoregressive technique to the neural network. The update is given as follows:

$$\mathbf{s}_t = 1/\sigma_t \text{ and } \mathbf{m}_t = -\mu_t/\sigma_t \text{ and } \sigma_t = \frac{1}{1+e^{-\mathbf{s}_t}} \text{ and } \mathbf{z}_t = \sigma_t \mathbf{z}_{t-1} + (1-\sigma_t)\mathbf{m}_t \tag{26}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ learnt from an Autoregressive neural network given in [6] receiving input $\mathbf{z}$ and $\mathbf{h}$ from the VAE.

**Sylvester Normalizing Flow** As explained earlier, planar flows suffer from the single-unit bottleneck problem. Sylvester Normalizing Flows (SNF) [1] attempt to solve this problem by modifying the transformation function which then behaves as an MLP with $M$ units instead of 1. SNF uses a transformation function of the following form

$$f(\mathbf{z}) = \mathbf{z} + \mathbf{A}h(\mathbf{B}\mathbf{z} + \mathbf{b}) \tag{27}$$

where $\mathbf{A} \in \mathbb{R}^{d\times m}, \mathbf{B} \in \mathbb{R}^{m\times d}$, $b \in \mathbb{R}^m$ with $m \leq d$, and $h$ is an element-wise non-linearity such as tanh. Using Sylvester's determinant identity, we can convert the computation of the determinant of a $d \times d$ matrix into the computation of the determinant of an $m \times m$ matrix.

$$\det(\mathbf{I}_d + \mathbf{A}\mathbf{B}) = \det(\mathbf{I}_m + \mathbf{B}\mathbf{A}) \tag{28}$$

Matrices $\mathbf{A}$ and $\mathbf{B}$ are further parameterized as $\mathbf{A} = \mathbf{Q}\mathbf{R}$ and $\mathbf{B} = \tilde{\mathbf{R}}\mathbf{Q}^\top$ where $\mathbf{R}$ and $\tilde{\mathbf{R}}$ are $m \times m$ upper-triangular matrices and $\mathbf{Q}$ is $d \times m$ matrix with orthonormal column vectors. The determinant of the Jacobian can then be written as

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \det\left(\mathbf{I}_m + \mathrm{diag}\left(h'(\tilde{\mathbf{R}}\mathbf{Q}^\top\mathbf{z} + \mathbf{b})\right)\tilde{\mathbf{R}}\mathbf{R}\right) \tag{29}$$

which can be computed in $O(m)$ time. Please refer to the original paper [1] for details.

### 4.2 Density Estimation

Density Estimation techniques take a different approach from Variational Inference (VI) methods to model the complex data distribution. Unlike VI, these methods aim for exact inference, sampling and log-likelihood evaluation. The primary goal in this regime is to find a bijective function $h = f(x)\ , x \in \mathcal{X}$ to map complex data-distribution $p_X(x)$ to density $p_H(f(x))$. Given that $p_H()$ has a simpler density whose likelihood function is analytically known, the overall log-likelihood of the data can be easily calculated. The complex log-likelihood of the data can now be calculated using the change of variables as follows:

$$\log p_X(x) = \log p_H(f(x)) + \log\left|\det\frac{\partial f(x)}{\partial x}\right| \tag{30}$$

**Non-linear Independent Components Estimation** Non-linear Independent Components Estimation (NICE) [3] is one of the early works adopting normalizing flows in density estimation. This work focuses on transformations $h = f(x)$ that maps the data into a factorized distribution, i.e., the components of $h_d$ are independent. Consequently, the log-likelihood in eq. (30) can be written as:

$$\log(p_X(x)) = \left[\sum_{d=1}^{D}\log p_{H_d}(f_d(x))\right] + \log\left|\det\frac{\partial f(x)}{\partial x}\right| \tag{31}$$

where $f(x) = (f_d(x))_{d \leq D}$.

This work targets invertible functions whose Jacobians have triangular structure so that calculating the determinant is tractable. In particular, it proposes the family of *coupling layers* that we define below.

*Coupling layer:* The coupling layer serves a building block of the transformation proposed in this work. The general coupling layer comprises of two partitions $I_1, I_2$ of the input dimensions $[1, D]$, such that $d = |I_1|$. The transformation is then defined as:

$$y_{I_1} = x_{I_1} \tag{32}$$

$$y_{I_2} = g(x_{I_2}; m(x_{I_1})) \tag{33}$$

where $g : \mathbb{R}^{D-d} \times m(\mathbb{R}^d) \rightarrow \mathbb{R}^{D-d}$ is an invertible function. Considering $I_1 = [1, d]$ and $I_2 = [d, D]$, the Jacobian of this function is:

$$\frac{\partial y}{\partial x} = \begin{bmatrix} I_d & 0 \\ \frac{\partial y_{I_2}}{\partial x_{I_1}} & \frac{\partial y_{I_2}}{\partial x_{I_2}} \end{bmatrix}$$

Where $I_d$ is the identity matrix of size $d$. That means that $\det \frac{\partial y}{\partial x} = \det \frac{\partial y_{I_2}}{\partial x_{I_2}}$ which evaluates to 1. The inverse of this transformation can be expressed as $x_{I_1} = y_{I_1}$ and $x_{I_2} = g^{-1}(y_{I_2}; m(y_{I_1}))$:

It is important to notice that the inverse of coupling function $m(.)$ is not required, thus allowing it to be modeled as complex non-linear functions. NICE adopts an *additive coupling law* which defines the function $g(.)$ as $g(a, b) = a+b$. Also, reLU is chosen as the coupling function $m(.)$.

A single coupling layer leaves part of the input unchanged. This is problematic, since modification of every dimension is desired. To achieve this, roles of partitions are interachanged in arjacent layers to ensure proper mixing.

**Real-valued Non-Volume Preserving** This subsequent work (RealNVP) [4] is an extension to the previous work (NICE) to enable multi-scalable architecture. The model uses *affine coupling law* as its primary constituent. This is defined as:

$$y_{1:d} = x_{1:d} \tag{34}$$

$$y_{d+1:D} = x_{d+1:D} \odot \exp\big(s(x_{1:d})\big) + t(x_{1:d}) \tag{35}$$

where $I_1 = [1, d]$ and $I_2 = [(d+1), D]$. Also, $\exp(s(.))$ represents the *scale* and $t(.)$ represents the *shift* operation respectively. The Jacobian of this transformation is also a triangular matrix (derivation has been provided in appendix).

Apart from the affine transformation, the model also proposes partitioning schemes across channels of input images and checkerboard pattern-based partitioning for exploiting pixel correlations. Further, to enable deep networks, the work proposes using techiniques such as Gaussianization and invertible *batch-normalization*. It achieves to get competitive scores against state-of-the variants, with some advantages that are discussed in the appendix.

**Masked Autoregressive Flow** The section on Inverse Autoregressive Flow (IAF) has already described how Autoregressive models such as MADE [6] can be used as normalizing flows. Masked Autoregressive Flow (MAF) [12] uses the transformation function in Eq. (21) contrary to IAF which uses the inverse function. Since the inverse function can be parallelized, MAF is suitable for computation of the density $p(\mathbf{x})$ of an externally provided data point $\mathbf{x}$. Sampling a new point from MAF requires $D$ sequential passes which cannot be parallelized and hence sampling is slow. However, in the case of IAF, as the inverse function is used, sampling is fast but density estimation is slow and requires $D$ sequential passes.

## 5 Normalizing Flows in Probabilistic Programming Languages

Normalizing flows have been implemented in two recent deep probabilistic programming languages Pyro (based on Pytorch) and Tensorflow Probability (TFP, based on Tensorflow). They provide a construct called the `Bijector` for the implementation of an invertible transformation required for construction of a normalizing flow. New transformation functions can be defined by extending the `Bijector` class. TFP also provides a construct called the `TransformedDistribution` which takes in a base distribution (which is generally a simple distribution) and a `Bijector`. A `TransformedDistribution` represents the distribution obtained after applying the `Bijector` on the base distribution. A number of functions commonly used in deep learning (e.g., Affine, Sigmoid, RealNVP) have been implemented as `Bijector`s in TFP. Another bijector called `Chain` is provided in TFP that converts a list of bijectors into a single bijector which represents a flow. The complete documentation of normalizing flows in TFP is given in [2].

Defining a custom transformation requires extending the `Bijector` class and implementing three functions: 1) `_forward` – receives samples $x$ from the base distribution as an input, evaluates the function $y = f(x)$, and returns the variable $y$. 2) `_inverse` – receives $y$ as the input and outputs $x$ corresponding to the inverse of the function, i.e., $x = f^{-1}(y)$. 3) `_inverse_log_det_jacobian` – receives $y$ as the input and computes the inverse log-determinant of the Jacobian, i.e., $\log\det\left|\frac{\partial f^{-1}}{\partial y}\right|$.

Let's define the three functions for an example function, the Parametrized ReLU (PReLU) function.

$$y = f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise, where } \alpha \in [0, 1] \end{cases} \tag{36}$$

$$x = f^{-1}(y) = \begin{cases} y & \text{if } y \geq 0 \\ \alpha^{-1} y & \text{otherwise} \end{cases} \tag{37}$$

$$\left.\frac{\partial f^{-1}}{\partial y}\right|_{ii} = \begin{cases} 1 & \text{if } y_i \geq 0 \\ \frac{1}{\alpha} & \text{otherwise} \end{cases} \tag{38}$$

An example bijector implementation for the PReLU function is given in the appendix.

## 6 Recent Advances

In this section, we discuss several recent works that used normalizing flows. We only explain the higher-level concept of each work. Interested readers should refer to the original paper given in the reference.

### 6.1 Pixel Recurrent Neural Network

Pixel Recurrent Neural Network (PixelRNN) [11] is an auto-regressive image generative model where the joint distribution over the image pixel is factorized into a product of conditional distribution. This means that the probability of pixel at position $i$ is given as: $p(x_i|x_1, \ldots, x_{i-1})$ where $x_1, \ldots, x_{i-1}$ is the previously generated pixels. This is a strong and counter-intuitive assumption for an image generation where pixel mostly conditioned on their neighbors. However, PixelRNN is proven to work well for image completion and generation task. In image completion, the occluded pixel is generated by conditioned upon the non-occluded pixels.

Oord et. al. [11] proposed three methods model an auto-regressive image generation. First, Row LSTM 1D convolution is used to generate image row by row from top to bottom. However, Row LSTM cannot capture the whole previously generated pixels since there is a coarse-graining in the convolution method. Second, Bidirectional LSTM is used to capture all the generated pixels as the context. Every pixel is conditioned upon their neighboring pixels except the one that has not been generated. However, LSTM training is known to be expensive so the authors proposed another model based on CNN. Third, Pixel CNN used a masked convolution by setting the filter of the pixels that has not been generated as zero.

### 6.2 Wavenet

Wavenet [14] used the idea of Pixel CNN from [11] for raw audio generation. It used one-dimensional convolution Pixel CNN to generate raw audio. Wavenet has been proven to be the state-of-the-art model for text-to-speech model. Google Assistant in Android use wavenet to generate the audio of the assistant.

Similar to Pixel CNN, the joint probability of an input audio is modelled by a stack of causal convolutional layers. Causal convolution has similar idea to masked convolution by shifting the output of a normal convolution by a few timesteps so it does not violate the autoregressive requirement. One of the problems with causal convolution is that it requires deep layers to capture long

range dependency. The authors proposed to use dilated convolution where the convolution filter is applied over an area larger than its length. This is done by skipping input values with a certain step.

### 6.3 Glow

Glow [8] extended and simplified the NICE and RealNVP model explained in Section 4.2 with three main extensions. First, the authors used activation normalization (act-norm) instead of batch normalization in RealNVP. Act-norm performs a channel-wise normalization and is faster than batch normalization. Second, the authors used $1 \times 1$ convolution for the channel-wise permutation. NICE and RealNVP proposed a flow containing the equivalent of a permutation that reverses the ordering of the channels. This is changed with a $1 \times 1$ convolution so that the permutation can be learned. Third, they extend RealNVP so that it can work faster by changing it into a multi-scale architecture. The authors splitted each step of the flow so it can be parallelized.

All of the additions mentioned before can be inverted and log-determinant of the Jacobian can be computed easily. Please refer to the paper for the details of the inversion and calculation of the Jacobian matrix. Glow has similar advantage as RealNVP that it can generate a high quality images. Moreover, the latent space learned by Glow is meaningful. This means that we can control the output of the generated image by tuning the latent space.

## 7 Conclusion

In this project, we have discussed how to transform a simple base distribution into a complex distribution by applying a series of invertible transformations called normalizing flows. We also discussed how normalizing flows can be applied to various representation learning regimes. Firstly, normalizing flows can be used for richer latent-posterior proposals for inference in the Variational Autoencoder. We discussed Planar and Radial Flows [13], Inverse Autoregressive Flow [9], and Sylvester Normalizing Flow [1] in this regime. Secondly, normalizing flows can be used to estimate density when the exact likelihood is not known. We discussed NICE [3], RealNVP [4], and Masked Autoregressive Flow [12] in this regime. We also discussed how normalizing flows can be implemented in deep probabilistic programming languages specifically Tensorflow Probability. Finally, we discussed several recent advances in normalizing flows used for image and audio generation.

In present day trends, applications of normalizing flows are being heavily used in methods involving variation inference. These are scalable methods which provide robust generative solutions. In contrast, auto-regressive methods suffer from slow sampling as the forward calculation of an autoregressive flow is not parallelizable. Density estimation methods ameliorate problems present in the mentioned methods, however their popularity remains constrained for historical reasons. Nevertheless, benchmark performance by models such as Glow seem to reverse this trend. From the PPL perspective, Tensorflow Probability provides

the highest range of bijector families, thus making it the ideal choice for flow-based implementations.

## References


1. van den Berg, R., Hasenclever, L., Tomczak, J.M., Welling, M.: Sylvester normalizing flows for variational inference. arXiv preprint arXiv:1803.05649 (2018)
2. Dillon, J.V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., Saurous, R.A.: Tensorflow distributions. arXiv preprint arXiv:1711.10604 (2017)
3. Dinh, L., Krueger, D., Bengio, Y.: Nice: Non-linear independent components estimation. arXiv preprint arXiv:1410.8516 (2014)
4. Dinh, L., Sohl-Dickstein, J., Bengio, S.: Density estimation using real nvp. arXiv preprint arXiv:1605.08803 (2016)
5. Figurnov, M., Mohamed, S., Mnih, A.: Implicit reparameterization gradients. arXiv preprint arXiv:1805.08498 (2018)
6. Germain, M., Gregor, K., Murray, I., Larochelle, H.: Made: Masked autoencoder for distribution estimation. In: International Conference on Machine Learning. pp. 881–889 (2015)
7. Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. science **313**(5786), 504–507 (2006)
8. Kingma, D.P., Dhariwal, P.: Glow: Generative flow with invertible 1x1 convolutions. arXiv preprint arXiv:1807.03039 (2018)
9. Kingma, D.P., Salimans, T., Jozefowicz, R., Chen, X., Sutskever, I., Welling, M.: Improved variational inference with inverse autoregressive flow. In: Advances in Neural Information Processing Systems. pp. 4743–4751 (2016)
10. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114 (2013)
11. Oord, A.v.d., Kalchbrenner, N., Kavukcuoglu, K.: Pixel recurrent neural networks. arXiv preprint arXiv:1601.06759 (2016)
12. Papamakarios, G., Murray, I., Pavlakou, T.: Masked autoregressive flow for density estimation. In: Advances in Neural Information Processing Systems. pp. 2338–2347 (2017)
13. Rezende, D., Mohamed, S.: Variational inference with normalizing flows. In: International Conference on Machine Learning. pp. 1530–1538 (2015)
14. Van Den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A.W., Kavukcuoglu, K.: Wavenet: A generative model for raw audio. In: SSW. p. 125 (2016)


## A Contribution

### Abdul Fatir Ansari

1. Introduction & Background (Autoencoder & Variational Autoencoder)
2. Applications
   - Variational Inference using Normalizing Flows (basics)
   - Planar and Radial Flow
   - Sylvester Normalizing Flow

   - Masked Autoregressive Flow
3. Normalizing flows in PPLs
4. Appendix: Implementation of planar flow with experiments on two complex densities
5. Problem Set 1
6. Code examples

**Devamanyu Hazarika**

1. Applications
   - Normalizing Flows for Density Estimation
   - Non-linear Independent Components Estimation
   - Real-valued Non-Volume Preserving
2. Conclusion
3. Appendix: Jacobian for affine transformation and discussion
4. Problem Set 2: Question 2

**Remmy A. M. Zen**

1. Applications
   - Inverse Autogregressive Flow
2. Normalizing flows in PPLs
3. Recent advances
4. Conclusion
5. Problem Set 2: Question 1

# B Appendix (Abdul Fatir Ansari)

## B.1 Planar and Radial Flow Derivations

*Planar Flow* Differentiating Eq. (15), the Jacobian is given by

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \mathbf{I} + \mathbf{u}h'(\mathbf{w}^\top \mathbf{z} + b)\mathbf{w}^\top$$

Now, using the matrix determinant lemma

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = (1 + h'(\mathbf{w}^\top \mathbf{z} + b)\mathbf{w}^\top \mathbf{I}^{-1}\mathbf{u}) \det(\mathbf{I}) \tag{39}$$

$$= (1 + h'(\mathbf{w}^\top \mathbf{z} + b)\mathbf{w}^\top \mathbf{u}) \tag{40}$$

*Radial Flow* Differentiating Eq. (17), the Jacobian is given by

$$\begin{aligned}\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} &= \mathbf{I} + \beta \left( (\mathbf{z} - \mathbf{z}_0) h'(\alpha, r) \frac{\partial r}{\partial \mathbf{z}} + h(\alpha, r)\mathbf{I} \right) \\ &= (1 + \beta h(\alpha, r))\mathbf{I} + \beta h'(\alpha, r)(\mathbf{z} - \mathbf{z}_0) \frac{(\mathbf{z} - \mathbf{z}_0)^\top}{||\mathbf{z} - \mathbf{z}_0||}\end{aligned}$$

Let $\gamma = (1 + \beta h(\alpha, r))$. Using the matrix determinant lemma

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \left( 1 + \beta h'(\alpha, r) \frac{(\mathbf{z} - \mathbf{z}_0)^\top}{||\mathbf{z} - \mathbf{z}_0||} \frac{\mathbf{I}}{\gamma} (\mathbf{z} - \mathbf{z}_0) \right) \det(\gamma \mathbf{I}) \tag{41}$$

$$= \left( \frac{1 + \beta h(\alpha, r) + \beta h'(\alpha, r)||\mathbf{z} - \mathbf{z}_0||}{(1 + \beta h(\alpha, r))} \right) (1 + \beta h(\alpha, r))^d \tag{42}$$

$$= (1 + \beta h(\alpha, r) + \beta h'(\alpha, r) r) \, (1 + \beta h(\alpha, r))^{d-1} \tag{43}$$

### B.2 Code for PReLU

```python
class PReLU(tfb.Bijector):
    def __init__(self, alpha=0.5, validate_args=False, name="p_relu"):
        super(PReLU, self).__init__(
            forward_min_event_ndims=0,
            validate_args=validate_args,
            name=name)
        self.alpha = alpha

    def _forward(self, x):
        return tf.where(tf.greater_equal(x, 0), x, self.alpha * x)

    def _inverse(self, y):
        return tf.where(tf.greater_equal(y, 0), y, 1. / self.alpha * y)

    def _inverse_log_det_jacobian(self, y):
        I = tf.ones_like(y)
        J_inv = tf.where(tf.greater_equal(y, 0), I, 1.0 / self.alpha * I)
        log_abs_det_J_inv = tf.log(tf.abs(J_inv))
        return log_abs_det_J_inv
```

Snippet 1.1: Parametrized ReLU Bijector

### B.3 Reproducing Results from Rezende and Mohamed [13]

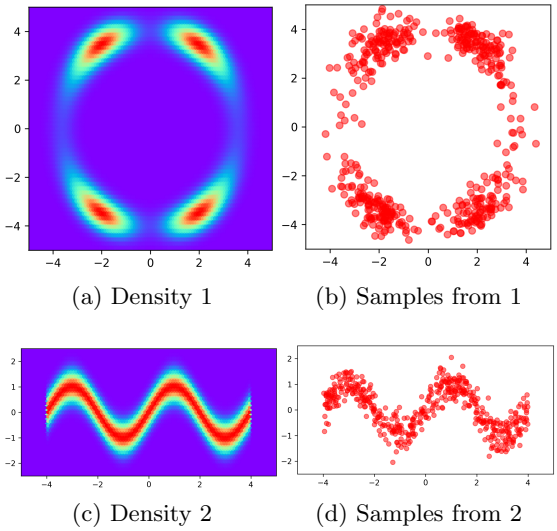
Reproducibility is an important concern in deep learning, especially because the models are trained stochastically in environments that can be very different

from each other. With this is mind, and a lack of official implementation from the authors, I implemented planar flows and conducted a few experiments presented in the paper.

The experiment seeks to approximate two complex 2D densities using planar flows. The unnormalized density functions are given by $p(\mathbf{z}) \propto e^{-U_i(\mathbf{z})}$ where the two energy functions $U_1$ and $U_2$ are

$$U_1(\mathbf{z}) = \frac{1}{2}\left(\frac{\|\mathbf{z}\| - 4}{0.4}\right)^2 - \log\left(e^{-\frac{1}{2}\left[\frac{z_1-2}{0.8}\right]^2} + e^{-\frac{1}{2}\left[\frac{z_1+2}{0.8}\right]^2}\right) \tag{44}$$

$$U_2(\mathbf{z}) = \frac{1}{2}\left[\frac{z_2 - \sin\left(\frac{2\pi z_1}{4}\right)}{0.4}\right]^2 \tag{45}$$



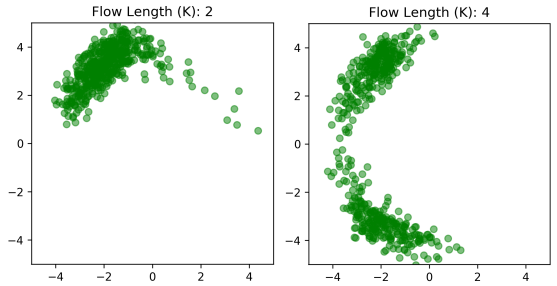
(a) Density 1 (b) Samples from 1

(c) Density 2 (d) Samples from 2

Fig. 3: Two probability densities and samples from each density obtained using Metropolis-Hastings

Fig. 3 shows the two densities along with a scatter plot of 500 samples from each density obtained using Metropolis-Hastings (MH) algorithm. The MH algorithm was implemented in `numpy` using the normal distribution with identity covariance matrix as the proposal distribution. A part of the code that implements the MH algorithm is given in Snippet 1.2. See `Metropolis-Hastings.ipynb` for complete code.

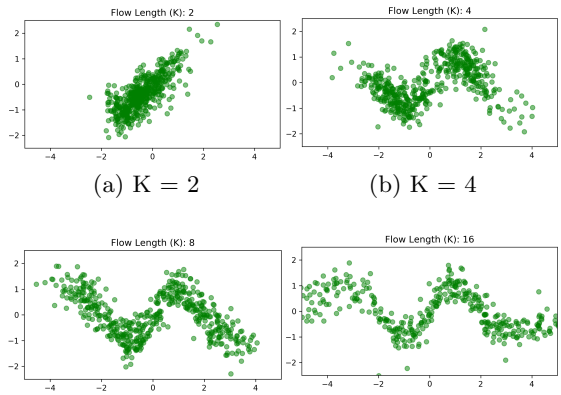
Planar Flow was implemented in Tensorflow and the code satisfies the invertibility conditions mentioned in the appendix of [13]. Tensorflow Probability (TFP) was not used because it requires implementation of the `_inverse` function




(a) K = 2 (b) K = 4

(c) K = 8 (d) K = 16

Fig. 4: 500 samples from Planar Flows with different flow lengths after 10000 training steps for density 1




(a) K = 2 (b) K = 4

(c) K = 8 (d) K = 16

Fig. 5: 500 samples from Planar Flows with different flow lengths after 5000 training steps for density 2

which is not available for the function used in planar flows although they are invertible. Flow lengths ($K$) from the set $\{2, 4, 8, 16\}$ were tested and trained using the Adam optimizer. A part of the code that implements planar flow is given in Snippet 1.3. See `PlanarFlow-Example1.ipynb` and `PlanarFlow-Example2.ipynb` for complete code. Note that constant terms were dropped during the computation of the KL-divergence which may lead to negative KL values.

Figs. 4 and 5 show samples from planar flows of different lengths for density 1 and density 2 respectively at the end of training. It is clear from the results that as flow length increases, the model is able to better approximate the true density. Another interesting aspect encountered during the course of experiments was that the training becomes unstable as length of flow $K$ increases, especially for density 2.

```
def metropolis_hastings(target_density, size=500000):
    burnin_size = 10000
    size += burnin_size
    x0 = np.array([[0, 0]])
    xt = x0
    samples = []
    for i in tqdm(range(size)):
        xt_candidate = np.array(
            [np.random.multivariate_normal(xt[0], np.eye(2))])
        accept_prob = (target_density(xt_candidate)) / (target_density(xt))
        if np.random.uniform(0, 1) < accept_prob:
            xt = xt_candidate
        samples.append(xt)
    samples = np.array(samples[burnin_size:])
    samples = np.reshape(samples, [samples.shape[0], 2])
    return samples
```

Snippet 1.2: Metropolis-Hastings

```
m = lambda x: -1 + tf.log(1 + tf.exp(x))
h = lambda x: tf.tanh(x)
h_prime = lambda x: 1 - tf.tanh(x) ** 2
base_dist = tfd.MultivariateNormalDiag(loc=[0., 0.], scale_diag=[1., 1.])
z_0 = base_dist.sample(500)
z_prev = z_0
sum_log_det_jacob = 0.
for i in range(K):
    with tf.variable_scope('layer_%d' % i):
        u = tf.get_variable('u', dtype=tf.float32, shape=(1, 2))
```

```
        w = tf.get_variable('w', dtype=tf.float32, shape=(1, 2))
        b = tf.get_variable('b', dtype=tf.float32, shape=())
        u_hat = (m(tf.tensordot(w, u, 2)) - tf.tensordot(w, u, 2)) * \
            (w / tf.norm(w)) + u
        affine = h_prime(tf.expand_dims(
            tf.reduce_sum(z_prev * w, -1), -1) + b) * w
        sum_log_det_jacob += tf.log(tf.abs(1 +
                                           tf.reduce_sum(affine * u_hat, -1)))
        z_prev = z_prev + u_hat * \
            h(tf.expand_dims(tf.reduce_sum(z_prev * w, -1), -1) + b)
z_k = z_prev
log_q_k = base_dist.log_prob(z_0) - sum_log_det_jacob
log_p = tf.log(true_density(z_k))

kl = tf.reduce_mean(log_q_k - log_p, -1)
```

Snippet 1.3: Planar Flow

## C Appendix (Devamanyu Hazarika)

First, let us derive the Jacobian for the affine coupling law defined in RealNVP:

$$\begin{aligned}\frac{\partial \mathbf{y}}{\partial \mathbf{x^T}} &= \begin{bmatrix} \frac{\partial y_{1:d}}{\partial x_{1:d}} & \frac{\partial y_{1:d}}{\partial x_{d+1:D}} \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}} & \frac{\partial y_{d+1:D}}{\partial x_{d+1:D}} \end{bmatrix} \\ &= \begin{bmatrix} I_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}} & \frac{\partial (x_{d+1:D} \odot \exp((s(x_{1:d}))) + t(x_{1:d}))}{\partial x_{d+1:D}} \end{bmatrix} \\ &= \begin{bmatrix} I_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}} & diag\,(\exp[s(x_{1:d})]) \end{bmatrix}\end{aligned}$$

Thus, the corresponding log determinant is:

$$\log\left(\left|\det(\frac{\partial \mathbf{y}}{\partial \mathbf{x^T}})\right|\right) = \sum_{i=1}^{d} \log(\exp(s(x_i)))$$

which can be calculated in $O(d)$ time.

*Discussion:* In their results, RealNVP doesn not surpass the state-of-the-art models. But it contains certain advantages over other proposed approaches. Firstly, compared to variational methods, density estimation methods provide tractable and exact log-lkelihood evaluation. This makes the model much more flexible. Secondly, there is no explicit likelihood function required with terms comprising reconstruction loss. This allows the generated images to be sharper

compared to other approaches such as variational autoencoders. Thirdly, unlike auto-regressive flows, the sampling scheme can be parallelized and thus modern hardware can be used to fasten the process.

When considering the discussed density estimation methods with other models such as VAEs and GANs, one key difference arises in the size of the latent space. RealNVP and similar techniques are able to find meaningful latent dimensions in a space with similar dimensions as the input. In contrast, other methods try to find meaningful latent states which are low dimensional. The former approach presents better expressibility and is thus desirable.