Normalizing Flows

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

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

Abstract. **Keywords:** Normalizing flows · .

1 Introduction

2 Background

2.1 Autoencoder

2.2 Variational Autoencoder

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.

$$\begin{aligned} F_Y(y) &= P(Y \leq y) && (1)\\ &= P(f(X) \leq y) && (2)\\ &= P(X \leq f^{-1}(y)) = F_X(f^{-1}(y)) && (3) \end{aligned}$$

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

$$F_Y(y) = F_X(f^{-1}(y)) = \int_{-\infty}^{f^{-1}(y)} p(x)dx \tag{4}$$

Differentiating Eq. (4) with respect to y (using the Fundamental Theorem of Calculus and the chain rule) we get

$$p(y) = p(f^{-1}(y)) \cdot \frac{df^{-1}}{dy} \tag{5}$$

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. (5) can be written as

$$p(y) = p(f^{-1}(y)) \cdot \left| \frac{df^{-1}}{dy} \right| \tag{6}$$

Eq. (6) can be extended to the multivariate case where the derivative is replaced by the determinant of the Jacobian matrix J

$$p(\mathbf{y}) = p(f^{-1}(\mathbf{y})) \cdot \left| \det \frac{\partial f^{-1}}{\partial \mathbf{y}} \right| = p(f^{-1}(\mathbf{y})) \cdot \left| \det \frac{\partial f}{\partial f^{-1}(\mathbf{y})} \right|^{-1} \tag{7}$$

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 $p(z|x)$ using a simpler variational family $q_\phi(z|x)$. 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. (7), the density function $q_K(\mathbf{z}_K)$ is given by

$$q_K(\mathbf{z}_K) = q_0(\mathbf{z}_0) \prod_{k=1}^{K} \left| \det \frac{\partial f_k}{\partial \mathbf{z}_{k-1}} \right|^{-1} \tag{8}$$

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

$$\mathcal{L} = \mathbb{E}_{q(\mathbf{z}_K|\mathbf{x})} \left[\log p_\theta(\mathbf{x}, \mathbf{z}_K) - \log q(\mathbf{z}_K|\mathbf{x}) \right] \tag{9}$$

$$= \mathbb{E}_{q(\mathbf{z}_0|\mathbf{x})} \left[\log p_\theta(\mathbf{x}, \mathbf{z}_K) - \log q(\mathbf{z}_K|\mathbf{x}) \right] \tag{10}$$

where $q(\mathbf{z}_0|\mathbf{x})$ is the simple initial density. Plugging in Eq. (8) into Eq. (10), we get a modified bound for flow-based VAEs

$$\mathcal{L} = \mathbb{E}_{q_0(\mathbf{z_0}|\mathbf{x})} \left[\log p(\mathbf{x}, \mathbf{z}_K) - \log q_0(\mathbf{z}_0|\mathbf{x}) + \sum_{k=1}^{K} \log \left| \det \frac{\partial f_k}{\partial \mathbf{z}_{k-1}} \right| \right] \tag{11}$$

Planar and Radial Flows Planar and Radial Flows [6] are one of the earliest flows proposed in the context of variational inference.

Planar flows use functions of the form

$$f(\mathbf{z}) = \mathbf{z} + \mathbf{u} h(\mathbf{w}^\top \mathbf{z} + b) \tag{12}$$

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 is then given by

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = (1 + h'(\mathbf{w}^\top \mathbf{z} + b)\mathbf{w}^\top \mathbf{u}) \tag{13}$$

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

Radial flows use functions of the form

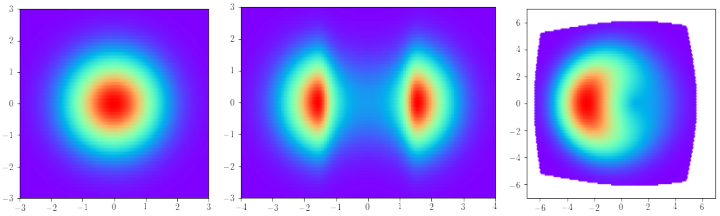
$$f(\mathbf{z}) = \mathbf{z} + \beta h(\alpha, r)(\mathbf{z} - \mathbf{z}_0) \tag{14}$$

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

The Jacobian is then given by

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = (1 + \beta h(\alpha, r) + \beta h'(\alpha, r) r) \left(1 + \beta h(\alpha, r)\right)^{d-1} \tag{15}$$

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



(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 Flows Planar and radial flow provide a simple invertible transformation shown to be effective in a low-dimensional latent space (up to hundred dimensions). Equation 12 of Planar Flow can be seen as Multilayer Perceptron with a single unit bottleneck hidden layer. This means that a long chain of transformations is needed to model a high-dimensional function.

Autoregressive flows [2] 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

$$p(\mathbf{y}) = \prod_{i=0}^{D} p(x_i | x_0, \ldots, x_{i-1}) \tag{16}$$

Kingma et. al. [4] proposed a Gaussian version of an autoregressive flow on a noise vector $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ given as follow:

$$y_0 = \mu_0 + \sigma_0 \epsilon_0 \tag{17}$$

$$y_i = \mu_i(\mathbf{y}_{0:i-1}) + \sigma(\mathbf{y}_{0:i-1})\epsilon_i \tag{18}$$

$$\tag{19}$$

This flow is invertible and the noise ϵ is given by:

$$\epsilon_i = \frac{y_i - \mu_i(\mathbf{y}_{0:i-1})}{\sigma(\mathbf{y}_{0:i-1})} \tag{20}$$

Note that, *epsilon* is independent to each other so the calculation of Equation 20 can be vectorized as follow:

$$\boldsymbol{\epsilon} = \frac{\mathbf{y} - \boldsymbol{\mu}(\mathbf{y})}{\boldsymbol{\sigma}(\mathbf{y})} \tag{21}$$

This enables an efficient computation with GPU.

Due to the autoregressive structure, the transformation has a lower triangular Jacobian where diagonal is σ_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:

$$\log\det\left|\frac{d\boldsymbol{\epsilon}}{d\mathbf{y}}\right| = \sum_{i=0}^{D} -\log\sigma_i(\mathbf{y}) \tag{22}$$

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.

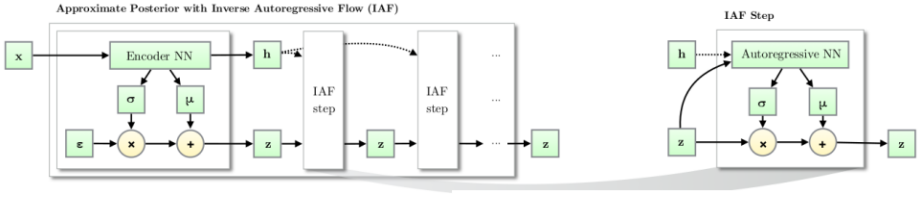


Fig. 2: The process of Inverse Autoregressive Flows in Variational Autoencoder [4].

The flow consists of a chain of T following transformations:

$$\mathbf{z}_t = \boldsymbol{\mu}_t + \boldsymbol{\sigma}_t \mathbf{z}_{t-1} \tag{23}$$

Kingma et.al. [4] 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:

$$\mathbf{s}_t = 1/\sigma_t \tag{24}$$
$$\mathbf{m}_t = -\mu_t/\sigma_t \tag{25}$$
$$\sigma_t = sigmoid(\mathbf{s}_t) \tag{26}$$
$$\mathbf{z}_t = \sigma_t \mathbf{z}_{t-1} + (1-\sigma_t)\mathbf{m}_t \tag{27}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ learnt from an Autoregressive neural network given in [2] receiving input $\mathbf{z}$ and $\mathbf{h}$ from the VAE.

4.2 Density Estimation

Non-linear Independent Components Estimation

Real-valued Non-Volume Preserving

5 Normalizing Flows in Probabilistic Programming Languages

Normalizing flows has been implemented in mainstream probabilistic programming languages that is based on mainstream deep learning framework such as Pyro (based on Pytorch) and Tensorflow Probability (TFP, based on Tensorflow). They provide a construct called the `Bijector` for the implementation of a function in normalizing flow. We can create our own normalizing function by extending the `Bijector` construct. TFP also provides a construct called the `TransformedDistribution` which takes two input a `Bijector` and the base distribution. `TransformedDistribution` will run the flow and transform the given base distribution with the given `Bijector`. There are already numerous function (e.g. Affine, Sigmoid, RealNVP) implemented as a `Bijector` in TFP. The complete documentation of normalizing flows in TFP is given in this reference [1].

To create your own function, you need to extend the `Bijector` construct and overload these three methods. First, the `_forward` function receives samples x from the base distribution as an input and calculate the value of the function $y = f(x)$. Second, the `_inverse` function areceives y from our function as the input nd output x corresponds to the inverse of the function $x = f^{-1}(y)$. Third, the `_inverse_log_det_jacobian` function receives y from our function as the input and calculate the inverse log-determinant of the Jacobian $\log\det\left|\frac{\partial f^{-1}}{\partial y}\right|$. After you implement your own function, you can create a list of bijectors corresponds to the flow. TFP provides another bijector called `Chain` that will chain your list of bijectors into a flow. You can then create an instance of `TransformedDistribution` with this chain and your base distributor.

6 Recent Advances

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

6.1 Pixel Recurrent Neural Network

Pixel Recurrent Neural Network (PixelRNN) [5] 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. [5] 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 [7] used the idea of Pixel CNN from [5] 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 [3] 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 1×1 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 1×1 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

References

1. 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)
2. Germain, M., Gregor, K., Murray, I., Larochelle, H.: Made: Masked autoencoder for distribution estimation. In: International Conference on Machine Learning. pp. 881–889 (2015)
3. Kingma, D.P., Dhariwal, P.: Glow: Generative flow with invertible 1x1 convolutions. arXiv preprint arXiv:1807.03039 (2018)
4. 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)
5. Oord, A.v.d., Kalchbrenner, N., Kavukcuoglu, K.: Pixel recurrent neural networks. arXiv preprint arXiv:1601.06759 (2016)
6. Rezende, D., Mohamed, S.: Variational inference with normalizing flows. In: International Conference on Machine Learning. pp. 1530–1538 (2015)
7. 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 Appendix (Contribution)

Abdul Fatir Ansari

1.

Devamanyu Hazarika

1.

Remmy A. M. Zen

1. Inverse autoregressive flows subsection.
2. Recent advances section.
3.

B Appendix (Abdul Fatir Ansari)

The Jacobian is then 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

$$\det \frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = (1 + h'(\mathbf{w}^\top \mathbf{z} + b)\mathbf{w}^\top \mathbf{I}^{-1}\mathbf{u}) \det(\mathbf{I}) \tag{28}$$

$$= (1 + h'(\mathbf{w}^\top \mathbf{z} + b)\mathbf{w}^\top \mathbf{u}) \tag{29}$$

C Appendix (Devamanyu Hazarika)

D Appendix (Remmy Zen)