Normalizing Flows

Muchang Bahng

Spring 2023

Contents

1 Normalizing Flows

We have seen many examples of generative models that attempt to produce a probability distribution p that approximates the true pdf p^* of the data samples. Some are given by an explicit model (e.g. GMMs, RBMs, VAEs) which approximate with parameterized form $p^* \sim p_\theta$, while in others (GANs) the model is implicit since we model the random variable as a transformation $X = G_{g_\theta}(Z)$ through a neural network with $Z \sim \mathcal{N}(0, I)$. If we focus on VAEs—specifically variational inference—for a second, recall that the performance of the model really just depends on the fact that we can approximate the intractable posterior $p(z \mid x)$ with some parameterized family $q_\phi(z)$[1]

$$\text{ELBO}(x^{(i)}, \phi, \theta) = \underbrace{\mathbb{E}_{q_\phi(z|x^{(i)})}[\log p_\theta(x^{(i)} \mid z)]}_{\substack{\text{likelihood term} \\ \text{(reconstruction part)}}} - \underbrace{KL(q_\phi(z \mid x^{(i)}) \mid\mid p(z))}_{\substack{\text{closeness of encoding to } p(z) \\ \text{(typically Gaussian)}}} \tag{1}$$

This may not be advantageous and flexible enough to capture the true posterior if $q_\phi(z)$ is far away from $p(z \mid x)$

To address this problem, in 2015, Google Deepmind through [RM16] introduced the idea of *flow-based* models [2] which—like GANs—want to map simple distributions (e.g. a Gaussian) to complex densities representing the data. This would normally result in an implicitly defined pdf, but by making the transformation to be *invertible*, we can use the change of basis formula to get a closed-form of the pdf, making flow models an *explicit* model of the pdf. Recall the lemma below from multivariate calculus.

Lemma 1.1 (Jacobi)

Let X, Z be absolutely continuous random variables in $\mathbb{R}^n$. Given that $f : \mathbb{R}^n \to \mathbb{R}^n$ is invertible and differentiable everywhere, with $X = f(Z), Z = f^{-1}(X)$, we claim

$$p_X(x) = p_Z(z) \cdot \left| \det \frac{\partial f^{-1}}{\partial x} \right| = p_Z(z) \cdot \left| \det \frac{\partial f}{\partial z} \right|^{-1} \tag{2}$$

where det is the determinant of the total derivative (Jacobian).

Proof.

For $n = 1$, we have

$$\begin{aligned} p_X(x) &= \frac{d}{dx} F_X(x) && (3) \\ &= \frac{d}{dx} F_Z(f^{-1}(x)) && (4) \\ &= p_z(f^{-1}(x)) \cdot \frac{d}{dx} f^{-1}(x) && (5) \\ &= p_z && (6) \end{aligned}$$

Therefore, if we parameterize f with some θ, then the marginal likelihood of x given θ can be written as

$$p_X(x;\ \theta) = p_Z(f_\theta^{-1}(x)) \cdot \left| \det \frac{\partial f_\theta^{-1}}{\partial x} \right| \tag{7}$$

Therefore, if X is a complex distribution and Z is a simple one (e.g. uniform), we might have hope to efficiently compute $p_X(x)$ for any $x \in \mathbb{R}^n$ if

[1] This can be written $q_\phi(z) = q_\phi(z \mid x)$, but since the encoder network produces the $\phi = E_\alpha(x)$, the $q_\phi(z)$ really represents a conditional distribution given x.

[2] This has nothing to do with flow graphs and the max-flow-min-cut theorem in graph theory.

1. we can efficiently compute $f_\theta^{-1}(x)$, which allows us to compute $p_Z(f^{-1}(x))$ efficiently since Z is simple.
2. we can efficiently compute the determinant of the Jacobian $|\det(\partial f_\theta^{-1}/\partial x)|$.[3]

1.1 Finite Normalizing Flows

It looks like we have split this enormously hard problem of modeling X with two slightly less difficult problems. However, with some tricks, we may be able to get a simple enough parameterization of f to be able to compute its inverse plus the determinant of the Jacobian. Note that the construction of a neural network was to take a simple function (one layer) and construct a complex family of functions composed of multiple layers. This is what we can do as well.

Corollary 1.1 (Finite Normalizing Flow)

Given a sequence of transformations

$$Z = Z_0 \xrightarrow{f_1} Z_1 \xrightarrow{f_2} \ldots \xrightarrow{f_{K-1}} Z_{K-1} \xrightarrow{f_K} Z_K = X \tag{8}$$

where each f_k is invertible and differentiable everywhere, let us denote $f = f_K \circ \ldots \circ f_1 : Z \to X$, which is invertible. Then

$$p_X(x) = p_Z(f^{-1}(x)) \cdot \prod_{k=1}^{K} \left| \det \frac{\partial f_k}{\partial z_{k-1}} \right|^{-1} \tag{9}$$

This sequence of random variables Z_k is called a **flow**, and the sequence of the corresponding pdfs p_{Z_k} is called the **normalizing flow**. As for how we parameterize this, our notation will assume that $\theta = (\theta_1, \ldots, \theta_M)$ and each θ_m parameterizes f_m.

Proof.

This can automatically be proved by induction. For an example, consider the sequence of invertible functions $Z \xrightarrow{f} Y \xrightarrow{g}\to X$, where Z is a simple distribution that gets transformed to a slightly more complicated distribution Y that then gets transformed to a complex distribution X. We can apply the Jacobi theorem above to see $p_Y(y) = p_Z(f^{-1}(y)) \cdot |(Df^{-1})(x)|$, and so we have

$$\begin{aligned} p_X(x) &= p_Y(g^{-1}(x)) \cdot |(Dg^{-1})(x)| && (10)\\ &= p_Z(f^{-1}(g^{-1}(x))) \cdot |(Df^{-1})(y)| \cdot |(Dg^{-1})(x)| && (11)\\ &= p_Z((g \circ f)^{-1}(x))) \cdot |(Df^{-1})(y)| \cdot |(Dg^{-1})(x)| && (12) \end{aligned}$$

Therefore, by taking a sequence these functions f_m, which may each have a simple parameterization, we may construct a very complex composition f_θ that may result in a very expressive Z. Again, this is very similar to how a composition of linear mappings plus an activation gives us a very expressive neural network, and unsurprisingly, there is an analogue of the universal approximation theorem for transformations of this form.

Theorem 1.1 (Probability Integral Transform)

Any n-dimensional random variable in $\mathbb{R}^n$ that is absolutely continuous w.r.t. the Lebesgue measure can be constructed from the uniform distribution U on $[0,1]^n$. Since we can map to and back from invertibility, any two such random variables X and Y can be mapped from each other,

$$X \mapsto U \mapsto Y \tag{13}$$

[3] Note that computing determinants are approximately $O(n^{2.4})$, which may not be practical.

Now given this, our job is to maximize the log-likelihood. Since $x = z_K$ and $z = z_0$, we substitute this in

$$\ln p_{Z_K}(x) = \ln p_{Z_0}(f^{-1}(x)) - \sum_{k=1}^{K} \ln \left| \det \frac{\partial f_k}{\partial z_{k-1}} \right| \tag{14}$$

and over the dataset we want to find

$$f^* = \underset{f}{\operatorname{argmax}} \sum_{i=1}^{N} \ln p_{Z_0}(f^{-1}(x^{(i)})) - \sum_{k=1}^{K} \ln \left| \det \frac{\partial f_k}{\partial z_{k-1}} \right| \tag{15}$$

where we are really optimizing over $f_1, \ldots, f_K$, i.e. their parameters $\theta_1, \ldots, \theta_K$. Great, now let's define the parametric form of f. There are two different types of invertible transformations that we can calculate in linear time.

Definition 1.1 (Planar Contractions)

Let us have $\theta = \{w \in \mathbb{R}^D, u \in \mathbb{R}^D, b \in \mathbb{R}\}$ and define the family of **planar contractions**

$$\{f(z) = z + u \cdot h(w^T z + b)\}_\theta \tag{16}$$

for some smooth nonlinearity h.[a] This is not always invertible, but the paper uses $h(x) = \tanh(x)$ with the fact that $w^T u \geq -1$ is sufficient for f to be invertible.

[a] As the paper states, this flow can be visualized as modifying the initial density by applying a series of contractions and expansions in the direction perpendicular to the hyperplane $w^T z + b$.

This already looks like a single layer, so you might wonder why one can't just directly make an invertible neural net? There is already literature on this, but the typical computational complexity of computing the determinant (plus the gradient of the determinant) is of scale $O(LD^3)$, where L is the number of hidden layers and D is the dimension of each hidden layer. It turns out that this decomposition gives us a very cute formula for computing determinants.

Lemma 1.2 ()

The determinant of a planar contraction is

$$\left| \det \frac{\partial f}{\partial z} \right| = \left| \det(I + u\psi(z)^T) \right| = \left| 1 + u^T \psi(z) \right| \tag{17}$$

where $\psi(z) = h'(w^T z + b) \cdot w$. This can be computed in $O(D)$ time.

Proof.

Use block matrix multiplication and the matrix determinant lemma.

The other type mentioned is a radial contraction.

Definition 1.2 (Radial Contraction)

Let us have $\theta = \{c \in \mathbb{R}^D, \alpha \in \mathbb{R}^+, \beta \in \mathbb{R}\}$ and define the family of **radial contractions**

$$\{f(z) = z + \beta h(\alpha, r)(z - c)\}_\theta \tag{18}$$

where $r = |z - c|$ and $h(\alpha, r) = \frac{1}{\alpha + r}$.[a]

[a] It applies radial contractions and expansions around the reference point c and are referred to as radial flows.

Lemma 1.3 ()

The determinant of a radial contraction is

$$\left| \det \frac{\partial f}{\partial z} \right| = \big[1 + \beta h(\alpha, r)\big]^{d-1} \big[1 + \beta h(\alpha, r) + \beta h'(\alpha, r) r\big] \tag{19}$$

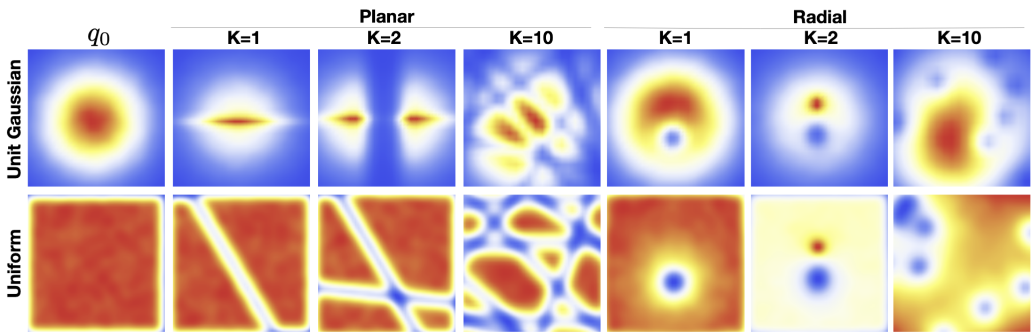


Figure 1: The effects of normalizing flow on two distributions. For example, for $k = 2$ we can transform a spherical Gaussian into a bimodal distribution by applying 2 successive transformations.

With this form, we can now substitute this into the variational lower bound. Note that we have not just simplified it with the following theorem, but we have also set it up as an expectation over the initial density $p(z)$, analogous to the reparamaterization trick that we have used for VAEs.

Theorem 1.2 ()

The variational lower bound can be written in the simplified form where the

$$\text{ELBO}(x) = \mathbb{E}_{q_0(z_0)}[\log q_0(z_0)] - \mathbb{E}_{q_0(z_0)}[\log p(x, z_K)] - \mathbb{E}_{q_0(z_0)}\left[\sum_{k=1}^{K} \log \left|1 + u_k^T \psi_k(z_{k-1})\right|\right] \tag{20}$$

for flow models.

Proof.

By definition $q_\phi(z \mid x) \coloneqq q_K(z_k)$, and substituting this in along with our simplified formula for the log likelihood gives us

$$\begin{aligned}
\text{ELBO}(x) &\coloneqq \mathbb{E}_{q_\phi(z|x)}\big[\log q_\phi(z \mid x) - \log p(x, z)\big] && (21)\\
&= \mathbb{E}_{q_K(z_K)}\big[\log q_K(z_K)\big] - \mathbb{E}_{q_K(z_K)}\big[\log p(x, z_K)\big] && (22)\\
&= \mathbb{E}_{q_K(z_K)}\left[\log q_0(z_0) - \sum_{k=1}^{K} \log \left|1 + u_k^T \psi_k(z_{k-1})\right|\right] - \mathbb{E}_{q_K(z_K)}\big[\log p(x, z_K)\big] && (23)
\end{aligned}$$

But since $z_K = f(z_0)$, using LOTUS we can get

$$\text{ELBO}(x) = \mathbb{E}_{q_0(z_0)}[\log q_0(z_0)] - \mathbb{E}_{q_0(z_0)}\left[\sum_{k=1}^{K} \log \left|1 + u_k^T \psi_k(z_{k-1})\right|\right] - \mathbb{E}_{q_0(z_0)}\big[\log p(x, z_K)\big] \tag{24}$$

At this point we can take the gradients, freely swap them with the expectations, and compute them.

1.2 Coupling-Layer Flows

This general method allowing linear-time computation of Jacobian determinants is an example of a **general normalizing flow**. Before, we have just constructed some family of invertible functions $\{f\}$ to transform the data, such that it was simple enough to calculate the Jacobian determinants in linear time. But note that in general, computing determinants requires you to have an triangular matrix (multiply all the diagonals). Therefore, if we can create a neural network f such that its Jacobian is triangular, we will be done. In order to have such an architecture to support this, we should introduce a special type of layer.

Definition 1.3 (Affine Coupling Layer)

Let us have an input $z \in \mathbb{R}^D$ such that it can be partitioned into its first d elements and the rest.

$$z = [z_{1:d}, z_{d+1:D}] \tag{25}$$

Now let's have 2 neural networks (not necessarily invertible) $F : \mathbb{R}^d \to \mathbb{R}^{D-d}$ and $H : \mathbb{R}^d \to \mathbb{R}^{D-d}$ defined

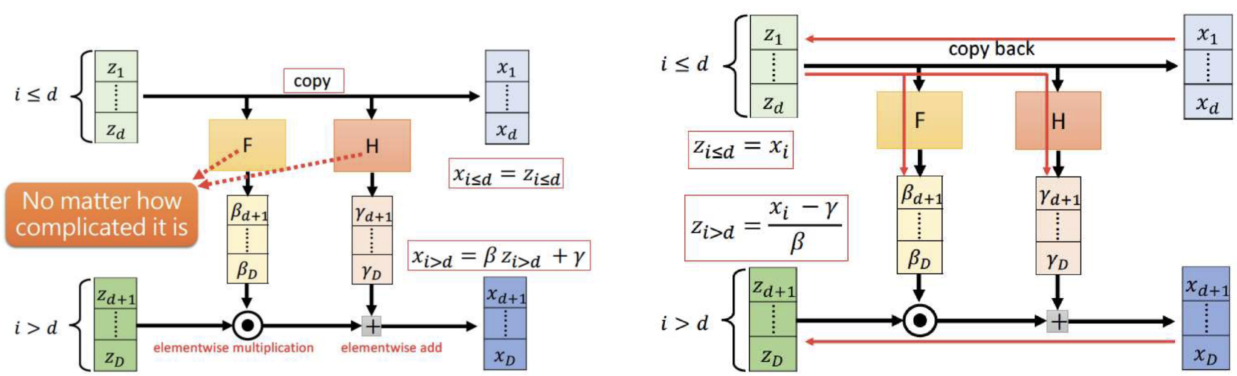
$$F(z') = \beta, \qquad G(z') = \gamma \tag{26}$$

Then the **coupling layer** $g : \mathbb{R}^D \to \mathbb{R}^D$ with parameters $\theta = (\theta_F, \theta_H)$ is defined as the transformation $x = g(z)$, where

$$x_{1:d} = z_{1:d} \tag{27}$$
$$x_{d+1:D} = z_{d+1:D} \odot \beta + \gamma = z_{d+1:D} \odot F(x_{1:d}) + H(x_{d+1:D}) \tag{28}$$

So this "layer" really composes of two neural networks, and it has both properties that we want. It is invertible, with inverse $z = g^{-1}(x)$ defined

$$z' = x', \qquad \bar{z} = \frac{\bar{z} - \gamma}{\beta} = \frac{\bar{z} - H(z')}{F(z')} \tag{29}$$



(a) Forward pass of the coupling layer.

(b) Backward pass of the coupling layer

Figure 2: Coupling layers are easily invertible. For both the forward and backward pass, we must do a forward pass through F and H.

Second, the determinant of this is easy to calculate since it is simply $\beta_1 \times \ldots \times \beta_{D-d}$. Therefore, we can stack these layers on top of each other, parameterized by different sequences of neural nets.

1.2.1 RealNVP

By setting each function f_i to be a cascading layer, we can model $f = f_K \circ \ldots \circ f_1$ and do the exact same normalizing flow model as mentioned before. This is pretty much NVP.

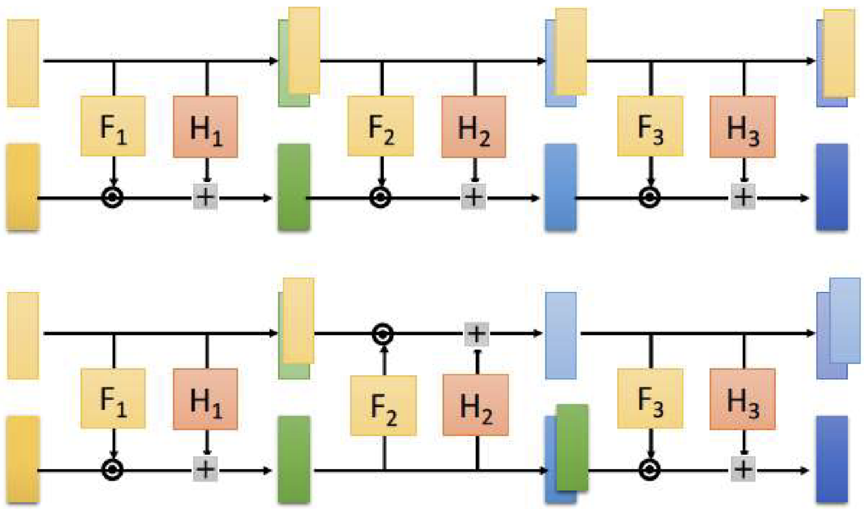


Figure 3: Cascading coupling layers represents f, where each layer is f_i.

1.2.2 NICE

Volume preserving flows design the flow such that the Jacobian determinant is equal to 1 (which is more restrictive but allows $O(1)$ computation) but still allows for rich posteriors. Therefore, even though computing the transformation $f_K \circ \ldots f_1$ will take longer due to the forward pass of neural nets, the lack of need to calculate the determinant keeps this fast. An example of such a volume-preserving coupling-layer flow model was introduced in [DKB15] by Dinh in 2014.

Definition 1.4 (NICE)

The **nonlinear independent components estimation (NICE)** model uses **additive coupling layers** of the form

$$x_{1:d} = z_{1:d} \tag{30}$$
$$x_{d+1:n} = z_{d+1:n} + H(x_{1:d}) \tag{31}$$

which we can see has a Jacobian determinant of one. The final layer of NICE is a multiplication by a diagonal matrix with all diagonal elements nonzeros, i.e. $x_i = \beta_i z_i$, which is invertible and has the absolute value of the determinant $\prod_i |\beta_i|$.

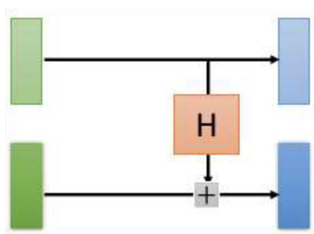


Figure 4: An additive coupling layer.

1.3 GLOW

Even with all these improvements with coupling-layers, flow models did not do as well as GANs. Therefore in 2018 Kingma, having invented the VAE, made a comeback by taking flow models and adding a lot of hacks in [KD18] to improve it, resulting in the GLOW model. There wans't really much theoretical backing to it, but it improved normalizing flow dramatically, finally resulting in a competitor for GANs.

Definition 1.5 (Flow Step in GLOW)

The GLOW layer consists of 3 components.

1. *Actnorm.*
2. *Invertible* 1×1 *Convolution.*
3. *Affine Coupling Layer.*

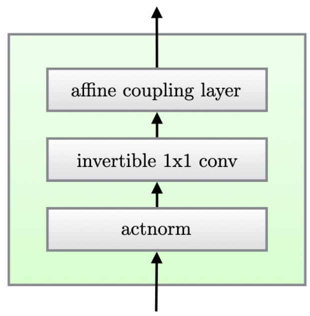


Figure 5: One step of GLOW.

After each layer, this is followed by a *squeeze* operation.

Definition 1.6 (Squeeze Operation)

For each channel, **squeezing** divides the image into sub-squares of shape $2 \times 2 \times c$, and then reshapes them into sub-squares of shape $1 \times 1 \times 4c$.

Definition 1.7 (Split and Concatenation Operation)

The **split operation** passes half of the input variables to further layers and shave off the other half as "finished." The **concatenation operation** performs the corresponding reverse operation: concatenation into a single tensor.

This results in the GLOW architecture.

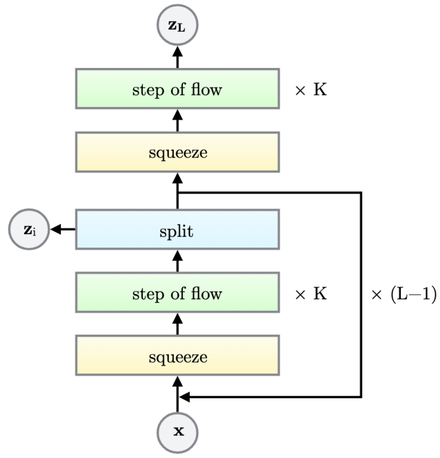


Figure 6: The complete glow architecture. $L = 6$ was known the be the best, but there's lots of fine-tuning.

You have a flow and squeeze a bunch of times is GLOW.

1.3.1 c-GLOW

There were further improvements by Lu in 2019, where in introduced conditional GLOW in [LH20]. It conditions on input noise, calculates the pdf of the noise, and subtracts it.

There are also applications in inpainting. You condition on the visible part and sample from conditional distribution.

1.4 Autoregressive Flows

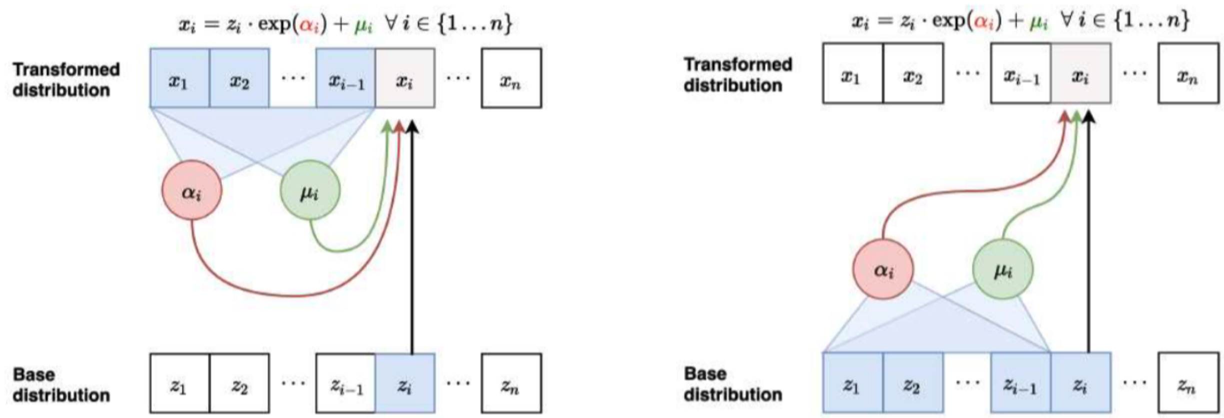
The nice form of flow models allows us to apply this to sequential data, i.e. in autoregressive models. Given words $x_1, \ldots, x_n$, sampled from $p(x_1, \ldots, x_n) = p(x_1)\, p(x_2 \mid x_1) \ldots p(x_n \mid x_1 \ldots x_{n-1})$, we would like to generate the next word. Doing so with RNNs can require lots of computation, so we can use *autoregressive models as flow models*. Consider a Gaussian nonlinear autoregressive model.

$$p(x) = \prod_{i=1}^{n} p(x_i \mid x_{<i}), \text{ where } p(x_i \mid x_{<i}) = \mathcal{N}\big(\mu(x_1, \ldots, x_{i-1}), \exp(\alpha_i(x_1, \ldots, x_{i-1}))^2\big) \tag{32}$$

where μ_i, α_i are deep neural networks (but not too deep due to computation, e.g. they can be fixed-length input resnets). Assuming that this is optimized somehow, we can sample from a Gaussian and run it through the neural nets to get the parameters of the next Gaussian distribution, sample them, and then do it again.

$$z_i \sim \mathcal{N}(0, 1), \qquad x_i = \mu_i + \exp(\alpha_i) z_i = \mu_i(x_{<i}) + \exp(\alpha_i(x_{<i})) z_i \tag{33}$$

This is the key idea to autoregressive normalizing flows. What we have just talked about is called a **masked autoregressive flow**.



(a) (b)

Figure 7

In an **inverse autoregressive flow**, the equations remain the same but the role of forward and backward passes have been reversed.

We can see a trade-off.

1. *Calculating log-likelihood.* MAFs are better since we can calculate it easy (remember we need determinant). IAFs requires us to go back and forth.
2. *Sampling.* IAFs are easier since we can sample from z_i's and sequentially calculate x_i's.

Can we get the benefits of both? Yes, and the general idea is to train the MAF first and then have the IAF student model imitate it.

Example 1.1 (Video Flows)

We can model a video as as a time series generated by a fancy random Gaussian walk, where the normal distributions are transformed by a neural network. This leads to *WaveNet*.

1.5 Infinitesimal Flows

1.6 Wasserstein Flows

References

[DKB15] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation, 2015.

[KD18] Diederik P. Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions, 2018.

[LH20] You Lu and Bert Huang. Structured output learning with conditional generative flows, 2020.

[RM16] Danilo Jimenez Rezende and Shakir Mohamed. Variational inference with normalizing flows, 2016.