

# Normalizing Flows

# 2

Draft version 2020-03-18 by Yang Yang and Jongha Jon Ryu

The basic idea underlying normalizing flow is to transform simple fixed distributions into flexible ones through learned differentiable and *invertible*<sup>1</sup> mappings. The simple distribution is often referred to as *base* distribution, and it has the following two properties:

1. It is easy to sample from.
2. Exact density can be evaluated for any sample given to it.

One example for such a distribution is multi-variant standard Gaussian: sampling is independent across different dimensions and is available in many scientific software libraries; its probability density function has a closed-form and can thus be easily evaluated.

Let  $\mathbf{Z}$  denote the random variable with the base distribution. A new random variable  $\mathbf{X}$  can be built by passing  $\mathbf{Z}$  through a deterministic function  $g_\theta$ , captured by a neural network with parameter  $\theta$ .

$$\mathbf{X} \triangleq g_\theta(\mathbf{Z}).$$

It turns out, if  $g_\theta$  is *differentiable and invertible*, both of the aforementioned properties carry over to  $\mathbf{X}$ . In other words, if we know how to sample from  $\mathbf{Z}$  and evaluate the exact density of  $\mathbf{Z}$ , then we can sample from and evaluate density for  $\mathbf{X}$  as well. We can think of this differentiable and invertible mapping as a bridge, which allows the two properties to be transferred from  $\mathbf{Z}$  to  $\mathbf{X}$ , and it is straightforward to chain multiple such bridges into a single one.

One important note is that  $\mathbf{X}$  and  $\mathbf{Z}$  must have the same dimension<sup>2</sup> for there to be a differentiable and invertible mapping between the two. In other words, there is no dimension reduction as we go from  $\mathbf{X}$  to  $\mathbf{Z}$  or the other way around.

In Section 2.1, we formalize the link between  $\mathbf{X}$  and  $\mathbf{Z}$  through the change of variables formula. In Section 2.2, we discuss two types of applications of normalizing flows: *density evaluation* and *sampling with density*, and make the observation that in either case we only need to evaluate the invertible mapping in one direction. In Section 2.3 and Section 2.4, we cover basic building blocks of normalizing flow networks and elements of network architecture design. Lastly we go over Neural ODE-based normalizing flows in Section 2.5 and summarize this chapter in Section 2.6.

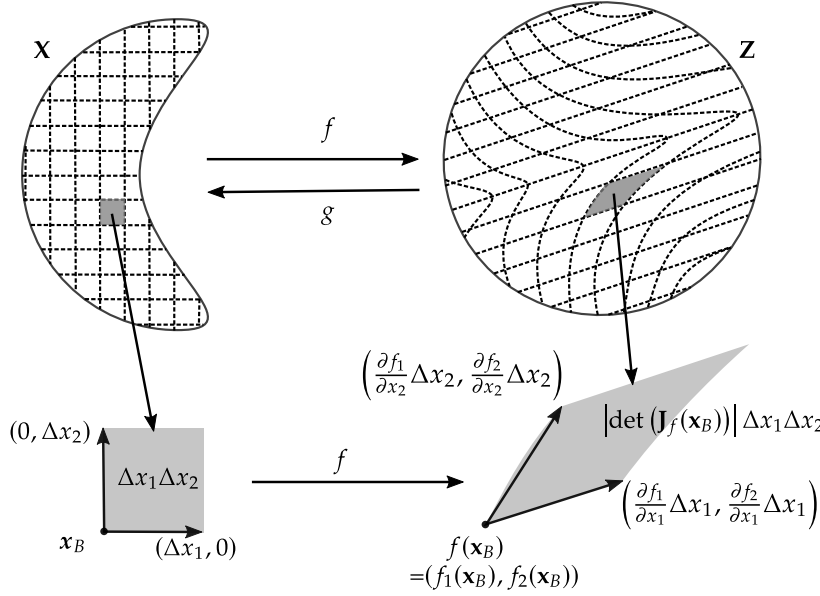
## 2.1 Change of variables

Let us define a multi-variant random variable  $\mathbf{X}$  as  $\mathbf{X} \triangleq g_\theta(\mathbf{Z})$ , where  $\mathbf{Z}$  is a multi-variant random variable with closed-form density function  $p_{\mathbf{Z}}$ ,

|   |    |
|---|----|
| 2.1 Change of variables . . . . .         | 11 |
| 2.2 Two classes of applications . . . . . | 14 |
| density evaluation . . . . .              | 14 |
| sampling with density . . . . .           | 15 |
| Summary . . . . .                         | 16 |
| 2.3 Basic building blocks . . . . .       | 17 |
| Affine coupling . . . . .                 | 18 |
| Auto-regressive flow . . . . .            | 19 |
| Planar flow* . . . . .                    | 21 |
| Residual flow* . . . . .                  | 23 |
| Comparison and remarks . . . . .          | 25 |
| 2.4 Network architectures . . . . .       | 26 |
| Multi-scale architecture . . . . .        | 27 |
| Other applications . . . . .              | 28 |
| Glow: a case study . . . . .              | 28 |
| 2.5 Continuous NFs . . . . .              | 31 |
| Neural ODE . . . . .                      | 31 |
| Change of variables for ODE . . . . .     | 32 |
| 2.6 Summary . . . . .                     | 33 |

1: Invertible mapping is also called bijective function, bijection, or one-to-one correspondence.

2: This is true only for continuous distributions, which is the focus of this chapter.



**Figure 2.1:** Geometric interpretation of  $|\det(\mathbf{J}_f(\mathbf{x}))|$ , the absolute value of the determinant of the Jacobian matrix: it represents a local and linearized rate of volume change around point  $\mathbf{x}$ .

and  $g_\theta$  is a differentiable and invertible mapping with inverse denoted as  $f_\theta \triangleq g_\theta^{-1}$ . The goal of this section to express the density of  $\mathbf{X}$  as a function of  $p_Z$  and  $g_\theta$ .

One invariance property as we link two random variables  $\mathbf{X}$  and  $\mathbf{Z}$  through an invertible mapping is that the probability mass of  $\mathbf{Z}$  in any area  $A$  must remain unchanged after the mapping, or more precisely,

$$\int_{g_\theta(A)} p_X(\mathbf{x}) d\mathbf{x} = \int_A p_Z(\mathbf{z}) d\mathbf{z}, \quad \text{or, equivalently,}$$

$$\int_B p_X(\mathbf{x}) d\mathbf{x} = \int_{f_\theta(B)} p_Z(\mathbf{z}) d\mathbf{z}, \quad \text{for any area } B. \quad (2.1)$$

To gain a geometric intuition, let us focus on the case when  $\mathbf{X}$  and  $\mathbf{Z}$  are two-dimensional. Consider the case when  $B$  is a rectangular area with one corner labeled as  $\mathbf{x}_B$  and two edges with length  $\Delta x_1$  and  $\Delta x_2$ , as illustrated in left half of Figure 2.1. Two approximations can be made when we  $\Delta x_1$  and  $\Delta x_2$  become infinitesimally small: (1) the left hand side of the above equation can be well approximated as  $p_X(\mathbf{x}_B)\Delta x_1\Delta x_2$ ; (2) the continuous function  $f_\theta(\mathbf{x})$  on the small rectangle can be well approximated by its first order Taylor expansion

$$f_\theta(\mathbf{x}) \approx f_\theta(\mathbf{x}_B) + \underbrace{\begin{bmatrix} \frac{\partial f_1(\mathbf{x}_B)}{\partial x_1} & \frac{\partial f_1(\mathbf{x}_B)}{\partial x_2} \\ \frac{\partial f_2(\mathbf{x}_B)}{\partial x_1} & \frac{\partial f_2(\mathbf{x}_B)}{\partial x_2} \end{bmatrix}}_{\mathbf{J}_f(\mathbf{x}_B)} (\mathbf{x} - \mathbf{x}_B).$$

The matrix in the equation above is the Jacobian<sup>3</sup> of function  $f_\theta$  evaluated at point  $\mathbf{x}_B$ , hereafter denoted as  $\mathbf{J}_f(\mathbf{x}_B)$ . Analogous to the interpretation of gradient in the case of a scalar-valued function, Jacobian represents the first-order linear and localized rate of change of a vector-valued function.

With this approximated linear transform, the rectangular area  $B$  is mapped to a parallelogram as illustrated in right part of Figure 2.1. One

3: The Jacobian matrix of a vector valued function  $f = [f_1, \dots, f_n]$  with input  $\mathbf{x} = [x_1, x_2, \dots, x_m]$  is defined as

$$\mathbf{J}_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix}.$$

As mentioned before, if  $f$  is an invertible function, its input and output should have the same dimension. Therefore in the case of normalizing flows, the Jacobian is always a square matrix. The matrix itself is a function of  $\mathbf{x}$ .

geometric property is that the area of the parallelogram equals to the absolute determinant<sup>4</sup> of the Jacobian matrix times the area of  $B$ :

$$\begin{aligned} \text{Area}(B) &= \Delta x_1 \Delta x_2 \\ \text{Area}(f(B)) &= |\det(\mathbf{J}_f(\mathbf{x}_B))| \Delta x_1 \Delta x_2. \end{aligned}$$

This allows us to rewrite Equation 2.1 as <sup>5</sup> :

$$\begin{aligned} \int_B p_{\mathbf{X}}(\mathbf{x}) d\mathbf{x} &= \int_{f_\theta(B)} p_{\mathbf{Z}}(\mathbf{z}) d\mathbf{z} \\ \stackrel{(a)}{\Rightarrow} p_{\mathbf{X}}(\mathbf{x}_B) \times \text{Area}(B) &= p_{\mathbf{Z}}(f(\mathbf{x}_B)) \times \text{Area}(f(B)) \\ \Rightarrow p_{\mathbf{X}}(\mathbf{x}_B) \Delta x_1 \Delta x_2 &= p_{\mathbf{Z}}(f(\mathbf{x}_B)) |\det(\mathbf{J}_f(\mathbf{x}_B))| \Delta x_1 \Delta x_2 \\ \Rightarrow p_{\mathbf{X}}(\mathbf{x}_B) &= p_{\mathbf{Z}}(f(\mathbf{x}_B)) |\det(\mathbf{J}_f(\mathbf{x}_B))|. \end{aligned}$$

Note how  $\Delta x_1 \Delta x_2$  cancels out and the final equation becomes an expression of density of  $\mathbf{x}_B$ . Since the choice of the rectangular area  $B$  and  $\mathbf{x}_B$  is arbitrary, this is the general rule, formally known as the the change of formula.

**Proposition 2.1.1** (Change of variable formula for probability density function) For  $\mathbf{X} \triangleq g(\mathbf{Z})$  where  $g(\cdot)$  is a differentiable invertible function with inverse  $f = g^{-1}$ , the density of  $\mathbf{X}$  can be expressed as

$$\begin{aligned} p_{\mathbf{X}}(\mathbf{x}) &= p_{\mathbf{Z}}(\mathbf{z}) |\det(\mathbf{J}_f(\mathbf{x}))|, \text{ or equivalently,} \\ p_{\mathbf{X}}(\mathbf{x}) &= p_{\mathbf{Z}}(\mathbf{z}) |\det(\mathbf{J}_g(\mathbf{z}))|^{-1}, \\ \text{where } \mathbf{x} &= g(\mathbf{z}), \mathbf{z} = f(\mathbf{x}). \end{aligned}$$

As illustrated in Figure 2.1,  $|\det(\mathbf{J}_f(\mathbf{x}))|$  represents a linearized rate of volume expansion around a local neighborhood of  $\mathbf{x}$ . With this interpretation in mind, the following two properties of determinant Jacobian should make intuitive sense, which we state without proof.

(1) The inverse of the determinant Jacobian is the determinant Jacobian of the inverse function.

$$\det(\mathbf{J}_f(\mathbf{x})) = \det(\mathbf{J}_g(\mathbf{z}))^{-1}, \text{ for } \mathbf{z} = f(\mathbf{x}), \mathbf{x} = g(\mathbf{z}).$$

This agrees with the intuition that the rate of localized volume *expansion* from  $\mathbf{x}$  to  $\mathbf{z} = f(\mathbf{x})$  should be the same the rate of localized volume *contraction* from  $\mathbf{z}$  to  $\mathbf{x} = g(\mathbf{z})$ .

(2) The determinant Jacobian of a composite function is the product of the determinant Jacobians of the composed functions.

$$\det(\mathbf{J}_{f \circ h}(\mathbf{x})) = \det(\mathbf{J}_f(\mathbf{y})) \times \det(\mathbf{J}_h(\mathbf{x})), \text{ for } \mathbf{y} = h(\mathbf{x}), \mathbf{z} = f(\mathbf{y})$$

Again, this is consistent with the intuition that rate of localized volume expansion is multiplicative.

As we will discuss in Section 2.3, the second property provides us a way to build complex distributions using simple non-linear invertible functions. It enables us to focus on the design of elementary parametrized invertible layers with tractable determinant Jacobian, knowing that more

4: Denoted as  $\det(\cdot)$

5: Step (a) holds in asymptotic sense when  $\Delta x_1$  and  $\Delta x_2$  approaches 0.

We motivated the change of variable equation with 2-dimensional random variable but the result generalizes to arbitrary dimension.

This follows by the fact that the matrix inverse of the Jacobian is the Jacobian of the inverse function.

$$\mathbf{J}_f(\mathbf{x}) = \mathbf{J}_g(\mathbf{z})^{-1}, \text{ for } \mathbf{z} = f(\mathbf{x}), \mathbf{x} = g(\mathbf{z}).$$

The inverse sign on the right hand side denotes a matrix inverse.

This follows from chain rule of Jacobian for multi-variant functions, a generalization of chain rule of derivative for scalar functions.

$$\mathbf{J}_{f \circ h}(\mathbf{x}) = \mathbf{J}_f(\mathbf{y}) \times \mathbf{J}_h(\mathbf{x})$$

The right hand side is a matrix multiplication.

expressive invertible transforms can be built by simply stacking these basic building blocks.

Before introducing basic building blocks in Section 2.3, we first need to understand what types of applications that normalizing flow are used for, as well as their implications on the design, which is covered next.

## 2.2 Two classes of applications

Not all invertible functions admit an explicit-form inverse. For instance, a simple scalar function  $f(x) = xe^x$  is invertible on  $\mathbb{R}^+$ , but it does not have an analytical inverse<sup>6</sup>. In other words, invertibility does not imply tractability. In the design of invertible neural network, we face the choice of whether to build invertible functions that are tractable in both directions, or to give up tractability in one direction in exchange for more flexibilities of the transform.

There is a fundamental trade-off between expressiveness and tractability in the design of the invertible functions used for normalizing flows. On the one hand, for efficient training or inference, it is desirable to have functions that are easy to compute, which demands tractability. On the other hand, we want to optimize a larger function class, which can be done by lifting the tractability constraint.

Fortunately, depending on the problem that we are interested in, we do not necessarily need to build functions that are tractable in both directions. Specifically, if we just want to use the model to evaluate density of a data sample  $\mathbf{x}$ , then we only need to evaluate  $f$  together with its determinant Jacobian. If instead we are interested in getting samples from the modeled distribution  $p_{\mathbf{X}}$  and know the density associated with the samples, then we only need to evaluate  $g$  together with its determinant Jacobian. We elaborate these two points in the following two subsections.

For notational clarity, for the rest of the chapter, we use subscript  $\theta$  in  $f_{\theta}, g_{\theta}$  to represent learnable parameters of the invertible functions, and replace  $p_{\mathbf{X}}$  with  $p_{\theta}$  to highlight the fact that the modeled distribution  $\mathbf{X}$  is parametrized by  $f_{\theta}$  or  $g_{\theta}$  through the change of variable formula.

### 2.2.1 density evaluation

In the problem of density evaluation, we are given a set of samples  $\{\mathbf{x}_i\}$  from some known distribution, and the task is to find the density of these samples with respect to the distribution captured by normalizing flow models. This problem arises in optimization problems that admit the following objective

$$\mathbb{E}_{\mathbf{x} \sim q_{\mathbf{X}}} [h(p_{\theta}(\mathbf{x}), \mathbf{x})], \tag{2.2}$$

where  $q_{\mathbf{X}}$  is given and there is known way to sample from it, and  $h(\cdot)$  denotes an arbitrary function. One common instance of such optimization

6: The inverse of  $f(x) = xe^x$  is known as Lambert W function, which cannot be expressed in terms of elementary functions.

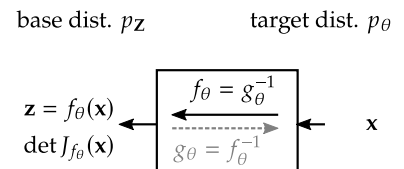


Figure 2.2: Density evaluation. There is no evaluation of  $g_{\theta}$  so it can be left implicit.

objective is the KL divergence from  $p_\theta$  relative to a fixed distribution  $q_X$ :

$$\begin{aligned} D_{\text{KL}}(q_X || p_\theta) &= \mathbb{E}_{\mathbf{x} \sim q_X} \left[ \log \frac{1}{p_\theta(\mathbf{x})} \right] - H(q_X) \\ &\approx \frac{1}{N} \sum_{\substack{i=1 \\ \mathbf{x}_i \sim q_X}}^N \log \frac{1}{p_\theta(\mathbf{x}_i)} + \text{Constant}. \end{aligned}$$

One distinct characteristic of this type of problem is that sampling of  $\mathbf{x}$  is driven by some other distribution (in the case of the above example,  $q_X$ ), and we just need to evaluate the log density of such samples with respect to the model distribution  $p_\theta$ . As a concrete example,  $q_X$  can be the empirical distribution of a dataset  $\{\mathbf{x}_i\}$ , in which case the above term is simply the negative log likelihood objective up to a constant. Minimization of the above term corresponds to maximum likelihood training.

By the change of variable formula in Proposition 2.1.1, we can express the density of  $p_\theta$  captured by the normalizing flow models as:<sup>7</sup>

$$p_\theta(\mathbf{x}) = p_Z(f_\theta(\mathbf{x})) |\det J_{f_\theta}(\mathbf{x})|, \quad (2.3)$$

which requires the computation  $f_\theta(\mathbf{x})$  and  $\det J_{f_\theta}(\mathbf{x})$ .

An important observation is that in this case, there is no need to evaluate  $g_\theta$ . In other words, we do not need to worry about the tractability of  $g_\theta$  if the objective is only to estimate the density of given samples.

## 2.2.2 sampling with density

In this second class of problems there are two tasks – we need to (1) draw samples  $\{\mathbf{x}_i\}$  from our modeled distribution  $p_\theta(\mathbf{x})$  and at the same time (2) obtain density of the drawn samples. The typical form of optimization objective is

$$\mathbb{E}_{\mathbf{x} \sim p_\theta} [h(p_\theta(\mathbf{x}), \mathbf{x})]. \quad (2.4)$$

Different from Equation 2.2, to evaluate the above term we need to sample from the modeled distribution  $p_\theta$ . One specific instance of the optimization objective is the KL divergence from  $q_X$  to  $p_\theta$  for some known distribution  $q_X$ :<sup>8</sup>

$$D_{\text{KL}}(p_\theta || q_X) = \mathbb{E}_{\mathbf{x} \sim p_\theta} \left[ \log \frac{p_\theta(\mathbf{x})}{q_X(\mathbf{x})} \right] \approx \frac{1}{N} \sum_{\substack{i=1 \\ \mathbf{x}_i \sim p_\theta}}^N \log \frac{p_\theta(\mathbf{x}_i)}{q_X(\mathbf{x}_i)}.$$

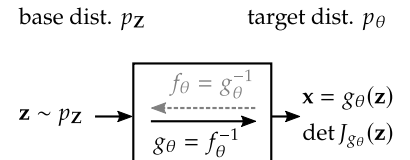
At first glance, it may appear that task (2) subsumes the first class of problems – to evaluate the density of the samples  $p_\theta(\mathbf{x}_i)$  inside the expectation, we need to invoke Equation 2.3 and map  $\{\mathbf{x}_i\}$  to  $\{\mathbf{z}_i\}$ , which necessitates evaluation of  $f_\theta$ . This, however, is not the case.

The subtlety here is that as long as the samples  $\{\mathbf{x}_i\}$  are generated by first sampling  $\{\mathbf{z}_i\}$  from  $p_Z$  and passing  $\{\mathbf{z}_i\}$  through  $g_\theta$ , their densities can be computed without evaluating  $f_\theta$ . To see this, let us rewrite Equation

$H(q_X) \triangleq -\int_{\mathbf{x}} q_X(\mathbf{x}) \log q_X(\mathbf{x})$  denotes the differential entropy of  $q_X$  and is a constant if  $q_X$  is a fixed distribution.

7: This is often written in log-density form as

$$\begin{aligned} \log p_\theta(\mathbf{x}) &= \log p_Z(f_\theta(\mathbf{x})) \\ &\quad + \log |\det J_{f_\theta}(\mathbf{x})| \end{aligned}$$



**Figure 2.3:** Sampling with density. There is no need to evaluate  $f_\theta$  so it can be left implicit.

8: E.g., in the case of variational autoencoder, to optimize evidence lower bound, we need to evaluate the KL divergence from prior distribution to approximated posterior distribution

$$D_{\text{KL}}(\text{approximated-posterior} || \text{prior})$$

and we can use normalizing flow to model the approximated posterior. Specifically, we can use a bijection where only  $g_\theta$  (mapping from base distribution to target distribution) is easy to compute [7, 8].

2.4 by explicitly expressing the sampling of  $\mathbf{x} \sim p_\theta$  as the sampling from the base distribution  $\mathbf{z} \sim p_Z$  followed by mapping through  $g_\theta$ <sup>9</sup>

$$\mathbb{E}_{\substack{\mathbf{z} \sim p_Z \\ \mathbf{x} = g_\theta(\mathbf{z})}} [h(p_\theta(\mathbf{x}))] = \mathbb{E}_{\mathbf{z} \sim p_Z} [h(p_\theta(g_\theta(\mathbf{z})))].$$

We know that  $p_\theta(g_\theta(\mathbf{z}))$  can be expressed as a function of  $p_Z(\mathbf{z})$  and  $\det J_{g_\theta}(\mathbf{z})$  through the change of variable formula in Proposition 2.1.1

$$p_\theta(g_\theta(\mathbf{z})) = p_Z(\mathbf{z}) |\det J_{g_\theta}(\mathbf{z})|^{-1},$$

which shows that the density of  $\mathbf{x} = g_\theta(\mathbf{z})$  can be computed by evaluating  $g_\theta$  and  $\det J_{g_\theta}$ .

Here is how we can interpret this equation. After obtaining a sample  $\mathbf{z}$  together its density  $p_Z(\mathbf{z})$  from the base distribution, we can map the sample as well as its density to the target domain with  $\mathbf{x} = g_\theta(\mathbf{z})$ . The density after the mapping needs to be adjusted taking into account the degree of contraction or expansion induced by  $g_\theta$  in a local neighborhood of  $\mathbf{z}$ , which is captured by  $|\det J_{g_\theta}(\mathbf{z})|$ . If  $|\det J_{g_\theta}(\mathbf{z})| > 1$ , then  $g_\theta$  maps a small volume around  $\mathbf{z}$  to a larger one around  $\mathbf{x}$ , stretching the density to be thinner. Conversely, if  $|\det J_{g_\theta}(\mathbf{z})| < 1$ , then the density get more concentrated through the mapping  $g_\theta$ .

Since there is no need to evaluate  $f_\theta = g^{-1}(\theta)$ , we do not need to have a tractable form of  $f_\theta$ .

### sample without density

As a degenerate scenario, we can also use normalizing flow models to obtain samples without the need to evaluate the density of such samples.

$$\mathbf{z} \sim P_Z, \mathbf{x} = g_\theta(\mathbf{z})$$

This, by itself, however, is not an interesting use-case for normalizing flows, since it does not require  $g_\theta$  to be invertible and we can instead use a much more flexible mapping. For example, in generative adversarial network (GAN) regime, a generator network maps samples from a base distribution to the target one. Since density evaluation is not needed, the generator network can be any flexible non-linear transform without invertibility as its design constraint.

That being said, for models trained with maximum likelihood, where at training time we need to evaluate  $f_\theta$  and its determinant Jacobian for density evaluation, sampling through  $g_\theta = f_\theta^{-1}$  is often performed at inference time as a way to showcase the quality of learned distribution. Since sampling is not done at training time, we can afford a  $g_\theta$  that is more computationally expensive to evaluate.

### 2.2.3 Summary

We summarize the two classes of applications and highlight two examples tasks: maximum likelihood density estimation and variational inference in Table 2.1.

9: Formally this is known as the law of the unconscious statistician or LOTUS

This is often written in log form as

$$\log p_\theta(g_\theta(\mathbf{z})) = \log p_Z(\mathbf{z}) - \log |\det J_{g_\theta}(\mathbf{z})|,$$

|                   | density evaluation   | sampling with density  |
|-------------------|--|--|
| Typical objective | $\mathbb{E}_{\mathbf{x} \sim q_{\mathbf{X}}} [h(p_{\theta}(\mathbf{x}), \mathbf{x})]$  | $\mathbb{E}_{\mathbf{x} \sim p_{\theta}} [h(p_{\theta}(\mathbf{x}), \mathbf{x})]$  |
| Need to evaluate  | $f_{\theta}$ and $\det J_{f_{\theta}}$   | $g_{\theta}$ and $\det J_{g_{\theta}}$   |
| Example           | Max-likelihood training<br>$\min_{\theta} D_{\text{KL}}(q_{\mathbf{X}} \  p_{\theta})$<br>$q_{\mathbf{X}}$ : empirical data dist.<br>$p_{\theta}$ : modeled data dist. | Variational inference<br>$\min_{\theta} D_{\text{KL}}(p_{\theta} \  q_{\mathbf{X}})$<br>$q_{\mathbf{X}}$ : prior dist.<br>$p_{\theta}$ : approx. posterior |

## 2.3 Basic building blocks

The main design goal in normalizing flows is to build flexible invertible mappings using deep neural networks, with the constraint that we need to be able to evaluate at least one direction of the bijective mapping as well as its determinant Jacobian.

Without loss of generality, we focus on the design of the mappings  $g_{\theta}$  from  $\mathbf{z}$  to  $\mathbf{x}$ , knowing that the same design can be used for  $f_{\theta}$  by just flipping the notation of input and output. More concretely, we look at designing neural network based function representations with input of  $\mathbf{z}$  and output of  $g_{\theta}(\mathbf{z})$  and  $\det J_{g_{\theta}}(\mathbf{z})$ . Since the modeled distribution is invariant to the sign of determinant Jacobian, it is a norm to design transform with strictly positive determinant Jacobian. Thus, subsequently, we remove the absolute sign.

The following two function composition properties allow us to break the problem of end-to-end network design down to the design of relative simpler invertible layers with limited expressive power, and rely on stacking multiple layers to achieve desired flexibility.

- a composite of bijections is also a bijection
- determinant Jacobian of a composite function is the product of the determinant Jacobians of the composed functions

For example, if we are given two invertible mappings  $g_{\theta_1}$  and  $g_{\theta_2}$ , we know that

$$g_{\theta_2} \circ g_{\theta_1} \triangleq g_{\theta_2}(g_{\theta_1}(\cdot)) \text{ is a bijection, and}$$

$$\log |\det J_{g_{\theta_2} \circ g_{\theta_1}}(\mathbf{z})| = \log |\det J_{g_{\theta_1}}(\mathbf{z})| + \log |\det J_{g_{\theta_2}}(g_{\theta_1}(\mathbf{z}))|.$$

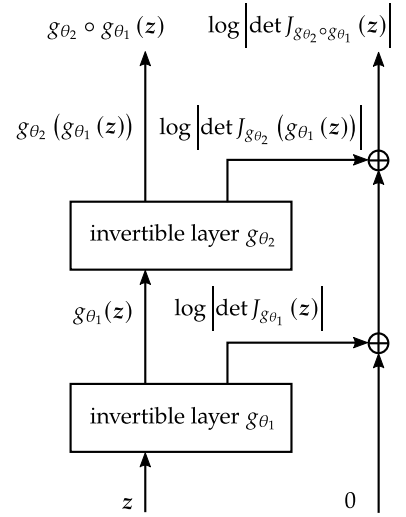
As illustrated in Figure 2.4, flexible and deep bijective networks can be constructed by stacking layers of basic invertible layers. In the subsequent sections, we introduce a few popular invertible layer designs and examine them in the following three aspects.

**Invertibility:** under what condition is the layer invertible.

**Inverse tractability:** how easy is it to evaluate the inverse function.

**Determinant Jacobian:** how to compute the determinant of Jacobian.

**Table 2.1:** Summary of two classes of applications for normalizing flow.  $p_{\theta}$  denotes the target distribution modeled by normalizing flow.  $q_{\mathbf{X}}$  denotes a known distribution.  $h(\cdot)$  denotes an arbitrary function.



**Figure 2.4:** Composition of two invertible layers

### 2.3.1 Affine coupling

In an affine coupling layer, input  $\mathbf{z}$  is split into two parts  $[\mathbf{z}_a, \mathbf{z}_b]$ : the first part is left unchanged  $\mathbf{x}_a = \mathbf{z}_a$  while the second part then goes through an affine transform  $\mathbf{x}_b = \mathbf{s} \odot \mathbf{z}_b + \mathbf{t}$  with the corresponding scaling  $\mathbf{s}$  and offset  $\mathbf{t}$  derived as a learned nonlinear function of the first part  $\mathbf{z}_a$ .

Since  $\mathbf{x}_a = \mathbf{z}_a$ , it is straightforward to see that in the inverse direction,  $\mathbf{s}$  and  $\mathbf{t}$  can be easily derived using the same nonlinear function as the forward pass. As long as  $\mathbf{s}$  is non-zero, the overall function is invertible, and  $\mathbf{z}_b$  can be obtained with the inverse of the affine transform. A typical design is to have  $\mathbf{s}$  as the output of activation function whose output is always positive, , such as exponential activation.

Below is the math description of an affine coupling layer. Figure 2.5 and Figure 2.6 illustrate the operation in the forward and inverse direction.

|  |  |
|--|--|
| <p>forward</p> $\mathbf{z}_a, \mathbf{z}_b = \text{split}(\mathbf{z})$ $\mathbf{t}, \log \mathbf{s} = \text{NN}_\theta(\mathbf{z}_a)$ $\mathbf{x}_a = \mathbf{z}_a$ $\mathbf{x}_b = \mathbf{s} \odot \mathbf{z}_b + \mathbf{t}$ $\mathbf{x} = \text{concat}(\mathbf{x}_a, \mathbf{x}_b)$ | <p>inverse</p> $\mathbf{x}_a, \mathbf{x}_b = \text{split}(\mathbf{x})$ $\mathbf{t}, \log \mathbf{s} = \text{NN}_\theta(\mathbf{x}_a)$ $\mathbf{z}_a = \mathbf{x}_a$ $\mathbf{z}_b = (\mathbf{x}_b - \mathbf{t})/\mathbf{s}$ $\mathbf{z} = \text{concat}(\mathbf{z}_a, \mathbf{z}_b)$ |
|--|--|

A desirable property of affine coupling is that its Jacobian matrix has a lower triangular form, as shown below.

$$J(\mathbf{z}) = \begin{bmatrix} \frac{\partial \mathbf{x}_a}{\partial \mathbf{z}_a} & \frac{\partial \mathbf{x}_a}{\partial \mathbf{z}_b} \\ \frac{\partial \mathbf{x}_b}{\partial \mathbf{z}_a} & \frac{\partial \mathbf{x}_b}{\partial \mathbf{z}_b} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \frac{\partial \mathbf{x}_b}{\partial \mathbf{z}_a} & \text{diag}(\mathbf{s}) \end{bmatrix}$$

This allows its determinant to be easily evaluated as the product of its diagonal terms. The off-diagonal term  $\frac{\partial \mathbf{x}_b}{\partial \mathbf{z}_a}$  does not affect the determinant of the matrix and thus can be ignored.

$$\log |\det J(\mathbf{z})| = \sum_i \log s_i.$$

Affine coupling is one instance of a general family of invertible transforms called coupling layers introduced in [9],

$$\begin{aligned} \mathbf{x}_a &= \mathbf{z}_a \\ \mathbf{x}_b &= h(\mathbf{z}_b, m(\mathbf{z}_a)) \end{aligned}$$

where  $h(\cdot)$  is an invertible mapping with respect to its first argument, referred to as the *coupling law* and  $m(\cdot)$  is a general function referred to as *coupling function*. Affine coupling applies affine transform as the coupling law hence the name. The degenerated case when the scaling of the affine coupling is fixed to be 1 is referred as *additive coupling*.

A limitation of coupling layer is that part of its input is left unchanged. To build flexible bijection we need to stack many of them together while shuffling the input elements so that elements that are not transformed in one layer can be modified in some other layer. Therefore, the stack of coupling layers is always interleaved with layers responsible for shuffling

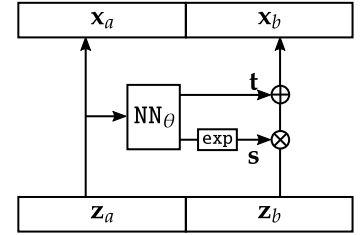


Figure 2.5: Affine coupling layer. Forward direction: from  $\mathbf{z}$  to  $\mathbf{x}$

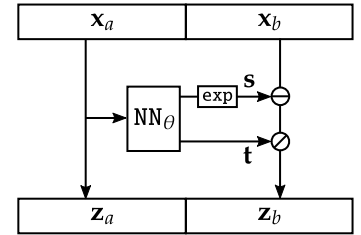


Figure 2.6: Affine coupling layer. Inverse direction: from  $\mathbf{x}$  to  $\mathbf{z}$

The determinant of a lower triangular matrix (or an upper triangular matrix) is the product of the diagonal entries.

[9]: Dinh et al. (2015), 'NICE: Non-linear Independent Components Estimation'

In case of additive coupling, the determinant Jacobian is fixed to be 1, implying that there is no volume expansion or contraction induced by the mapping. Transforms with this property are referred as being *volume-preserving*.



of input elements. The pattern at which the input elements can be shuffled is an important design element of coupling layer based normalizing flow networks.

In the first two initial works that applies affine coupling [9, 10] (NICE and RealNVP),  $[z_a, z_b]$  are alternated in between successive layers. [10] proposes two additional approaches to split input tensor: spatial checkerboard partitioning and channel-wise partitioning. [11] generalizes the fixed input permutation to a learned 1x1 convolution, and use it together with a fixed channel-wise partitioning method. We will go through the network architecture of [11] in more details in Section 2.4.

The properties of affine coupling are summarized in the table below.

|                             |  |
|-----------------------------|--|
| Condition for invertibility | Invertible as long as $s_i \neq 0, \forall i$ , which is enforced by using an activation function with strictly positive outcome, e.g., $\exp$ |
| Tractability of inverse     | forward and inverse can be evaluated with the same computational complexity  |
| Log determinant Jacobian    | Jacobian matrix is lower-triangular, and $\log \det J_{g_\theta} = \sum_i \log s_i$  |

Table 2.2: Summary of affine coupling

### 2.3.2 Auto-regressive flow

Auto-regressive flow defines a mapping where each input element  $z_i$  goes through an affine transform whose scale  $s_i$  and offset  $t_i$  are derived as a learned function of previous elements  $\mathbf{z}_{<i}$ . It can be expressed as

$$\mathbf{x} = \text{AF}(\mathbf{z}) \Leftrightarrow \begin{cases} \mathbf{x} = \mathbf{t} + \mathbf{s} \odot \mathbf{z} \text{ (i.e., } x_i = t_i + s_i * z_i, \forall i), \\ \mathbf{t}, \log \mathbf{s} = \text{AutoregressiveNN}_\theta(\mathbf{z}). \end{cases}$$

We use `AutoregressiveNN` to denote a neural network that takes in a sequence of data as input and generate a sequence of outputs, with the property that the  $i^{\text{th}}$  output element does not depend on  $i^{\text{th}}$  input or any input that appears later in the sequence. If the input sequence is indexed by time, then `AutoregressiveNN` is strictly causal – current output only depends on previous inputs.

As expressed below, we can view `AutoregressiveNN` as a collection of networks  $[\text{NN}_\theta^{(i)}]_i$ , potentially sharing parameters, that can be executed in parallel. This is illustrated in Figure 2.7.

$$\left. \begin{array}{l} t_1, \log s_1 = \text{constant} \\ t_2, \log s_2 = \text{NN}_\theta^{(1)}(\mathbf{z}_{\leq 1}) \\ \dots \\ t_i, \log s_i = \text{NN}_\theta^{(i-1)}(\mathbf{z}_{\leq i-1}) \\ \dots \end{array} \right\} \Leftrightarrow \mathbf{t}, \log \mathbf{s} = \text{AutoregressiveNN}_\theta(\mathbf{z}) \quad (2.5)$$

As a consequence of the strict causality property, both  $\partial t_i / \partial z_j$  and  $\partial s_i / \partial z_j$

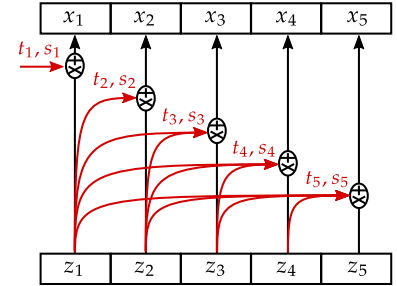


Figure 2.7: Autoregressive Flow (forward). Red arrows represent learned functions parametrized by neural networks.

are 0 whenever  $i \leq j$ , which further implies that  $\partial x_i / \partial z_j = 0$  for  $i < j$ :

$$x_i = t_i + s_i z_i,$$

$$\Rightarrow \frac{\partial x_i}{\partial z_j} = \underbrace{\frac{\partial t_i}{\partial z_j}}_{=0 \text{ for } i \leq j} + \underbrace{\frac{\partial s_i}{\partial z_j}}_{=0 \text{ for } i \leq j} z_i + \underbrace{\frac{\partial z_i}{\partial z_j}}_{\substack{=0 \text{ for } i \neq j \\ =1 \text{ for } i=j}} s_i = \begin{cases} 0 & i < j \\ s_i & i = j \end{cases}.$$

This leads to a lower diagonal Jacobian matrix  $J = \partial \mathbf{x} / \partial \mathbf{z}$  with easy to compute determinant:

$$J(\mathbf{z}) = \begin{bmatrix} s_1 & 0 & \dots & 0 & 0 \\ \frac{\partial x_2}{\partial z_1} & s_2 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \frac{\partial x_{N-1}}{\partial z_1} & \frac{\partial x_{N-1}}{\partial z_2} & \dots & s_{N-1} & 0 \\ \frac{\partial x_N}{\partial z_1} & \frac{\partial x_N}{\partial z_2} & \dots & \frac{\partial x_N}{\partial z_{N-1}} & s_N \end{bmatrix}$$

$$\log |\det J(\mathbf{z})| = \sum_i \log s_i$$

### Connection to affine coupling layer

A closer inspection of Equation 2.5 and Figure 2.7 reveals that autoregressive flow can be viewed as a composition of  $N$  affine coupling layers that transform one element of the input at a time. Specifically, the first layer splits input  $\mathbf{z}$  unevenly into  $\mathbf{z}_{\leq N-1}$  and  $z_N$ , leave  $\mathbf{z}_{\leq N-1}$  unchanged, and transform  $z_N$  into  $x_N$ ; next layer repeats this process by focusing on  $\mathbf{z}_{\leq N-1}$ . In general, we can express the operation of one of the coupling layers as:

$$\begin{aligned} \mathbf{z}_{\leq i-1}, z_i &= \text{split}(\mathbf{z}_{\leq i}) \\ t_i, \log s_i &= \text{NN}_{\theta}^{(i-1)}(\mathbf{z}_{\leq i-1}) \\ x_i &= s_i z_i + t_i \\ \text{output} &= \text{concat}(\mathbf{z}_{\leq i-1}, x_i) \end{aligned}$$

Since the inputs of these affine coupling layers do not depend on the output of one another, they can be executed in parallel. This parallelism, however, comes at a cost of a slow serialized inverse.

### Inverse of autoregressive flow

While its forward direction can be computed with a single execution of  $\text{AutoregressiveNN}_{\theta}$ , the inverse of an autoregressive flow is much slower to evaluate.

$$z_i = (x_i - t_i) / s_i$$

Since  $t_i$  and  $s_i$  depends on  $\mathbf{z}_{\leq i-1}$ , we need to first obtain  $\mathbf{z}_{\leq i-1}$  before obtaining  $z_i$ . To obtain all  $N$  elements of  $\mathbf{z}$ , we need to compute  $\text{NN}_{\theta}^1, \text{NN}_{\theta}^2, \dots$  sequentially in a serialized fashion, where each evaluation of  $\text{NN}_{\theta}^i$  yields only one element of  $\mathbf{z}$ . The inverse operation is depicted in Figure 2.8.

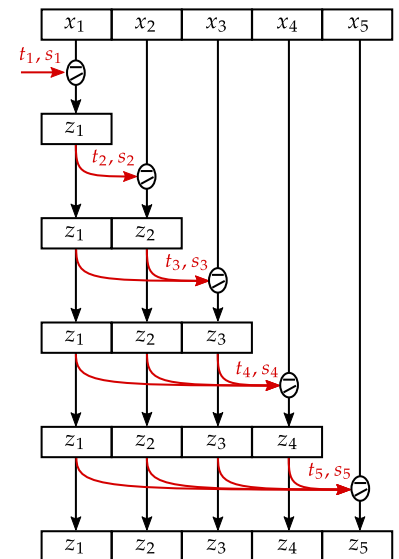


Figure 2.8: Autoregressive Flow (inverse). Red arrows represent learned functions parametrized by neural networks.

This sequential procedure of the inverse can be prohibitively slow for problems with very large data dimension.

### Summary of autoregressive flow

The properties of autoregressive flow are summarized in the table below.

|                             |  |
|-----------------------------|--|
| Condition for invertibility | Invertible as long as $s_i \neq 0, \forall i$  |
| Tractability of inverse     | Inverse is tractable but requires slow sequential evaluation. Forward pass, in contrast, can be evaluated in parallel. |
| Log determinant Jacobian    | Jacobian matrix is lower-triangular, so $\log \det J_{g_\theta} = \sum_i \log s_i$                                     |

**Table 2.3:** Summary of autoregressive flow

In this section we introduced autoregressive flow as a mapping from  $\mathbf{z}$  to  $\mathbf{x}$ , but it can be used to map from  $\mathbf{x}$  to  $\mathbf{z}$  as well. Since the inverse of autoregressive flow is much more costly to compute compared with its forward pass, we need to carefully choose which direction to apply it to, depending on whether we want to favor *density evaluation* or *sampling with density*. For instance, if we intend to build a model for maximum likelihood training, then we can use autoregressive flow with  $\mathbf{x}$  as input to allow efficient parallel density estimation during training. If instead the goal is to efficiently sample from a modeled distribution and optionally evaluate the modeled density of such samples, then it is sensible to use autoregressive flow in the direction from  $\mathbf{z}$  to  $\mathbf{x}$ . This design trade-off is highlighted in Table 2.4 below.

|                         | $\mathbf{x} = \text{AF}_\theta(\mathbf{z})$ | $\mathbf{z} = \text{AF}_\theta(\mathbf{x})$ |
|-------------------------|---|---|
| density estimation      | sequential                                  | parallel                                    |
| sampling (with density) | parallel                                    | sequential                                  |
| References              | [7]   | [12]  |

**Table 2.4:** Comparison of using autoregressive flow to map from  $\mathbf{z}$  to  $\mathbf{x}$ , and from  $\mathbf{x}$  to  $\mathbf{z}$ .

In [7], the case of  $\mathbf{x} = \text{AF}_\theta(\mathbf{z})$  is branded as *Inverse Auto-regressive Flow (IAF)*, and used to model approximated posterior in VAE. This is not to be confused with the inverse of auto-regressive flow transform.

So far we have assumed the existence of an `AutoregressiveNNθ` without discussing its detailed network form. The design of `AutoregressiveNN` is an important topic in itself and warrants a separate chapter on its own. Properly designed, even a single layer of autoregressive transform can be a quite powerful model. We will dive into more details in Chapter 3.

### 2.3.3 Planar flow\*

Planar flow is introduced in [8] and defined with the following form:

$$\mathbf{x} = \mathbf{z} + \mathbf{u}h(\mathbf{w}^T \mathbf{z} + b). \quad (2.6)$$

$h(\cdot)$  is a nonlinear *scalar* activation function (e.g.,  $\tanh$ ), and  $\theta \triangleq \{\mathbf{u}, \mathbf{w}, b\}$  is the set of learnable parameters. This transform shifts every point  $\mathbf{z}$  in the direction of  $\mathbf{u}$ , by the amount determined by the projection of  $\mathbf{z}$  on  $\mathbf{w}$ . In the degenerated case when  $h(\cdot)$  is an identity function, planar flow can be viewed as a combination of shear and scaling transform, as is illustrated in the middle figure of Figure 2.9, where scaling happens along the direction of  $\mathbf{w}$  and shear is applied in the subspace perpendicular to  $\mathbf{w}$ . A nonlinear function  $h(\cdot)$  then applies non-linear contraction or expansion along direction of  $\mathbf{w}$ . In other words, the change in volume happens perpendicular to the hyper-plane of  $\mathbf{w}^T \mathbf{z} + b = 0$ , which explains the name *planar flow*.

The Jacobian of planar flow can be derived as

$$\begin{aligned} J(\mathbf{z}) &= \frac{\partial \mathbf{x}}{\partial \mathbf{z}} = \frac{\partial \mathbf{z}}{\partial \mathbf{z}} + \mathbf{u} \frac{\partial h(\mathbf{w}^T \mathbf{z} + b)}{\partial \mathbf{z}} \\ &= \frac{\partial \mathbf{z}}{\partial \mathbf{z}} + \mathbf{u} \frac{\partial h(\mathbf{w}^T \mathbf{z} + b)}{\partial (\mathbf{w}^T \mathbf{z} + b)} \frac{\partial (\mathbf{w}^T \mathbf{z} + b)}{\partial \mathbf{z}} \\ &= \mathbf{I} + \underbrace{h'(\mathbf{w}^T \mathbf{z} + b)}_{\text{scalar}} \underbrace{\mathbf{u} \mathbf{w}^T}_{\text{rank-1 matrix}}. \end{aligned}$$

The above form is referred to as rank-1 perturbation of identity matrix, and from *matrix determinant lemma*, its determinant has a closed form below

$$\begin{aligned} \det J(\mathbf{z}) &= \det (\mathbf{I} + h'(\mathbf{w}^T \mathbf{z} + b) \mathbf{u} \mathbf{w}^T) \\ &= 1 + h'(\mathbf{w}^T \mathbf{z} + b) \mathbf{u}^T \mathbf{w}. \end{aligned}$$

It can be shown that planar flow is invertible as long as the above term is positive for any  $\mathbf{z}$ , i.e.,

$$\mathbf{u}^T \mathbf{w} > -\frac{1}{h'(\mathbf{w}^T \mathbf{z} + b)}.$$

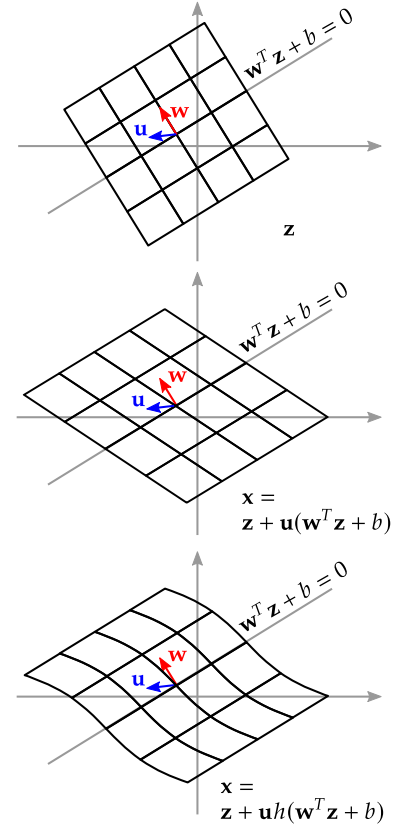
For typical activation function such as  $h = \tanh$  with maximum derivative of 1, the above condition can be met with  $\mathbf{u}^T \mathbf{w} > -1$ , which can be enforced with some reparameterization trick.

There are two major limitations of planar flow that prevent it from being widely applicable: (1) There is no known closed-form inverse. Finding the inverse requires numerical iteration that is much more inefficient to evaluate compared to forward pass. (2) More importantly, the expressivity of a single layer planar flow is severely limited – there are only two vectors and one scalar  $\{\mathbf{u}, \mathbf{w}, b\}$  as its learnable parameter, the nonlinearity of the entire layer is driven by a single neuron, and it only expands or contracts space orthogonal to a single hyper-plane.

### Sylvester flow as an extension

To remedy the second limitation, [13] proposes an extended version of planar flow named Sylvester flow with the following form

$$\mathbf{x} = \mathbf{z} + \mathbf{U}h(\mathbf{W}^T \mathbf{z} + \mathbf{b}).$$



**Figure 2.9:** Planar flow in 2 dimensional space. Top: original  $\mathbf{z}$  space. Middle: planar flow with identity  $h(\cdot)$  is similar to shear transform; Bottom: with nonlinear  $h(\cdot)$ , planar flow expands or contracts space perpendicular to  $\mathbf{w}^T \mathbf{z} + b = 0$ .

#### matrix determinant lemma

$$\det (\mathbf{I} + \mathbf{u} \mathbf{w}^T) = 1 + \mathbf{u}^T \mathbf{w}$$

Note that enforcing  $J(\mathbf{z}) > 0$  only guarantees local invertibility (by *inverse function theorem*), but not necessarily global invertibility.

In this specific case, though, it can be shown that  $J(\mathbf{z}) > 0$  leads to global invertibility. For details please refer to Appendix A.1. in [8].

[13]: Berg et al. (2019), *Sylvester Normalizing Flows for Variational Inference*

Here  $\mathbf{U}$  and  $\mathbf{W}$  are learnable matrices with dimension  $N \times M$ ,  $\mathbf{b}$  is a learnable vector with dimension  $M$ , and  $h(\cdot)$  denotes an activation function applied element-wise on its input. It removes the single-neuron limitation by applying  $M$  non-linear activations per layer instead of one. The name originates from the use of *Sylvester determinant identity* in the derivation of its determinant Jacobian

$$\begin{aligned} J(\mathbf{z}) &= \frac{\partial \mathbf{z}}{\partial \mathbf{z}} + \mathbf{U} \frac{\partial h(\mathbf{W}^T \mathbf{z} + \mathbf{b})}{\partial (\mathbf{W}^T \mathbf{z} + \mathbf{b})} \frac{\partial (\mathbf{W}^T \mathbf{z} + \mathbf{b})}{\partial \mathbf{z}} \\ &= \mathbf{I}_{N \times N} + \mathbf{U} \text{diag} (h'(\mathbf{W}^T \mathbf{z} + \mathbf{b})) \mathbf{W}^T, \\ \det J(\mathbf{z}) &= \det (\mathbf{I}_{M \times M} + \text{diag} (h'(\mathbf{W}^T \mathbf{z} + \mathbf{b})) \mathbf{W}^T \mathbf{U}). \end{aligned}$$

To ease computation of the above,  $\mathbf{W}$  and  $\mathbf{U}$  are parametrized as  $\mathbf{QR}$  and  $\mathbf{Q}\tilde{\mathbf{R}}^T$  respectively, where the columns in  $\mathbf{Q}$  are a set of orthonormal vectors, and  $\mathbf{R}$  and  $\tilde{\mathbf{R}}$  are  $M \times M$  upper-triangular matrices. This reduces determinant Jacobian to be  $\det J(\mathbf{z}) = \prod_i (1 + h'_i(\mathbf{W}^T \mathbf{z} + \mathbf{b}) r_{ii} \tilde{r}_{ii})$ , where  $r_{ii}$  and  $\tilde{r}_{ii}$  are  $i^{\text{th}}$  diagonal entry of  $\mathbf{R}$  and  $\tilde{\mathbf{R}}$ . The main design component in Sylvester flow is the parametrization and learning of the orthogonal matrix  $\mathbf{Q}$ , which we leave out and refer interested readers to Section 3.2 in [13].

### Summary of planar flow

The properties of planar flow are summarized in the table below.

|                             |   |
|-----------------------------|---|
| Condition for invertibility | $\mathbf{u}^T \mathbf{w} > -1$ when $\tanh$ is used as activation function.   |
| Tractability of inverse     | Inverse is not tractable.   |
| Determinant Jacobian        | Jacobian matrix has the form of rank-1 perturbation of the identity and its determinant is derived from matrix determinant lemma. |

#### Sylvester determinant identity

$$\det (\mathbf{I}_{N \times N} + \mathbf{U} \mathbf{W}^T) = \det (\mathbf{I}_{M \times M} + \mathbf{W}^T \mathbf{U}),$$

where  $\mathbf{U}$  and  $\mathbf{W}$  are  $N \times M$  matrices.

Invertibility is guaranteed when  $(1 + h'_i(\mathbf{W}^T \mathbf{z} + \mathbf{b}) r_{ii} \tilde{r}_{ii}) > 0$  for all  $i$  [13].

Table 2.5: Summary of planar flow

### 2.3.4 Residual flow\*

Residual flow or invertible residual layer [14, 15] is defined in the same way as a typical residual layer [16]

$$\mathbf{x} = \mathbf{z} + r_\theta(\mathbf{z}) \quad (2.7)$$

which is the sum of a skip connection  $\mathbf{z}$  and a learned residual module  $r_\theta(\mathbf{z})$ . Next we show that a sufficient condition for the residual layer to be invertible is that  $r_\theta$  is Lipschitz continuous with the Lipschitz constant, denoted as  $\text{Lip}(r_\theta)$ , less than 1.

**Proposition 2.3.1**  $\mathbf{x} = \mathbf{z} + r_\theta(\mathbf{z})$  is invertible if  $r_\theta(\mathbf{z})$  is Lipschitz continuous and the Lipschitz constant of the residual module  $r_\theta$ ,  $\text{Lip}(r_\theta)$ , is less than 1.

*Proof.* For any given  $\mathbf{x}$ , let us define a mapping  $T_{\mathbf{x}}(\mathbf{z}) \triangleq \mathbf{x} - r_{\theta}(\mathbf{z})$ , then we have

$$\begin{aligned} \|T_{\mathbf{x}}(\mathbf{z}_1) - T_{\mathbf{x}}(\mathbf{z}_2)\|_2 &= \|r_{\theta}(\mathbf{z}_1) - r_{\theta}(\mathbf{z}_2)\|_2 \\ (r_{\theta} \text{ is Lipschitz continuous}) \quad &\leq \text{Lip}(r_{\theta}) \|\mathbf{z}_1 - \mathbf{z}_2\|_2 \end{aligned}$$

Since  $\text{Lip}(r_{\theta}) < 1$ , the above inequality shows that  $T_{\mathbf{x}}$  is a *contraction mapping*. By *Banach fixed-point theorem*, we know that there is a unique fixed point  $\mathbf{z}^*$  satisfying

$$T_{\mathbf{x}}(\mathbf{z}^*) = \mathbf{z}^*.$$

In other words, for any  $\mathbf{x}$ , there is a unique  $\mathbf{z}^*$  such that  $\mathbf{x} = \mathbf{z}^* + r_{\theta}(\mathbf{z}^*)$ .  $\square$

*Banach fixed-point theorem* additionally shows that such a fixed point  $\mathbf{z}^*$  can be found by the following iterative procedure with any arbitrary  $\mathbf{z}^{(0)}$ ,

$$\begin{aligned} \mathbf{z}^{(n)} &= T_{\mathbf{x}}(\mathbf{z}^{(n-1)}) = \mathbf{x} - r_{\theta}(\mathbf{z}^{(n-1)}), \\ \mathbf{z}^{(n)} &\xrightarrow{n \rightarrow \infty} \mathbf{z}^*. \end{aligned}$$

Unlike previously introduced invertible layers that rely on certain structures of Jacobian matrix for the efficient evaluation of its determinant, the Jacobian of a residual layer takes a general form below

$$J(\mathbf{z}) = \frac{\partial \mathbf{z}}{\partial \mathbf{z}} + \frac{\partial r_{\theta}(\mathbf{z})}{\mathbf{z}} = \mathbf{I} + J_r(\mathbf{z}),$$

where  $J_r$  denotes the Jacobian of the residual module  $r_{\theta}$ . To compute its log determinant, we resort to the following matrix identity

$$\log \det \mathbf{A} = \text{tr} \log \mathbf{A}$$

where the matrix logarithm is defined as

$$\log(\mathbf{I} + \mathbf{A}) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\mathbf{A}^k}{k}.$$

By combining the above three equations, we can express the log determinant into an infinite sum

$$\log \det J = \text{tr} \log(\mathbf{I} + J_r) = \text{tr} \left( \sum_{k=1}^{\infty} (-1)^{k+1} \frac{J_r^k}{k} \right) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\text{tr}(J_r^k)}{k}. \quad (2.8)$$

A truncated sum with finite terms can be used as a biased approximation [14] of the true log determinant Jacobian, or an unbiased estimation can be obtained using a Russian roulette estimator [15].

### Enforcement of Lipschitz constraint

From Proposition 2.3.1 we know that ensuring invertibility translates to enforcing Lipschitz constraint on the residual module  $r_{\theta}$ . Specifically, we want to be able to train  $r_{\theta}$  with some optimization objective while at this

$\log \det \mathbf{A} = \text{tr} \log \mathbf{A}$  is the matrix version of  $\log \prod_i a_i = \sum \log a_i$ .

The trace  $\text{tr}$  is a linear operator, hence it commutes with summation  $\Sigma$ .

[14]: Behrmann et al. (2019), *Invertible Residual Networks*

same time make sure that its Lipschitz constant is strictly less than 1. The immediate next question is how to enforce such constraint.

First we should note that in general the residual module is a multi-layer network, which can be expressed as a sequence of function composition  $r_\theta = r^{(1)} \circ r^{(2)} \circ \dots \circ r^{(l)} \circ \dots$  where  $r^{(i)}$  is either a non-linear activation function or a linear operation such as convolution or fully connected layer. It can be easily verified that the Lipschitz constant of  $r_\theta$  is upper bounded by the product of the Lipschitz constant of each individual layer, that is

$$\text{Lip}(r_\theta) < \prod_l \text{Lip}(r^{(l)}).$$

Therefore,  $\text{Lip}(r_\theta) < 1$  can be achieved by enforcing  $\text{Lip}(r^{(l)}) \leq 1$  for each individual layer and  $\text{Lip}(r^{(l)}) < 1$  for at least one layer.

For most element-wise activation functions (ReLU, tanh, ELU, Sigmoid) that have a maximum derivative of 1, its Lipschitz constant is 1.<sup>10</sup> As non-element-wise non-linearity, softmax is known to be Lipschitz 1 [17], and it is easy to show that the same is true for max-pool.

Any linear operation can be expressed as a matrix multiplication  $f(\mathbf{x}) = \mathbf{W}\mathbf{x}$ . As we can see below, the spectral norm of  $\mathbf{W}$ , i.e., the largest singular value of  $\mathbf{W}$ , is the smallest Lipschitz constant of  $f$ .

$$\begin{aligned} \|\mathbf{W}\|_2 &\triangleq \max_{\mathbf{x} \neq 0} \frac{\|\mathbf{W}\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \max_{\mathbf{x}, \mathbf{y}, \mathbf{x} \neq \mathbf{y}} \frac{\|\mathbf{W}(\mathbf{x} - \mathbf{y})\|_2}{\|\mathbf{x} - \mathbf{y}\|_2} \\ \Rightarrow \|\mathbf{W}\mathbf{x} - \mathbf{W}\mathbf{y}\|_2 &\leq \|\mathbf{W}\|_2 \|\mathbf{x} - \mathbf{y}\|_2 \end{aligned}$$

Thus, we reduce the problem of ensuring invertibility of the residual layer as the problem of regularizing the spectral norm of linear operations in residual module  $r_\theta$ . We end the discussion by noting that constraining the spectral norm of linear layers is a useful technique that is needed under many contexts, and refer interested readers to [18–20].

### Summary of residual flow

|                             |  |
|-----------------------------|--|
| Condition for invertibility | $r_\theta$ is Lipschitz continuous with Lipschitz constant less than 1   |
| Tractability of inverse     | There is no closed-form inverse. Inverse can be obtained through a fixed point iteration.  |
| Determinant Jacobian        | log determinant Jacobian can be expressed as an infinite sum involving Jacobian of $r_\theta$ (Equation 2.8) and approximated with a finite sum. |

10: If an element-wise activation function  $a$  has maximum derivative of 1, then it has Lipschitz constant of 1 with arbitrary length vector at any p-norm.

$$\begin{aligned} &\|a(\mathbf{x}) - a(\mathbf{y})\|_p \\ &= \left( \sum_i (a(x_i) - a(y_i))^p \right)^{1/p} \\ &\leq \left( \sum_i (x_i - y_i)^p \right)^{1/p} \\ &= \|\mathbf{x} - \mathbf{y}\|_p \end{aligned}$$

Table 2.6: Summary of residual flow

### 2.3.5 Comparison and remarks





So far we have gone over four popular invertible layers in normalizing flows, and detailed their characteristics in terms of condition for invertibility, tractability of inverse and the derivation of determinant Jacobian matrix. Their comparison is summarized in Table 2.7.

One thing to stress is that even though the layers are introduced as a mapping from  $\mathbf{z}$ , the base distribution, to  $\mathbf{x}$ , the target distribution, it is merely a choice of exposition. They can very well be applied from  $\mathbf{x}$  to  $\mathbf{z}$  as well.

Note, however, that except for affine coupling which is symmetric in its forward and inverse in terms of computational complexity, the rest three have clear polarity in terms of forward and inverse tractability or computational complexity. Specifically, the inverse of autoregressive flow is much slower to evaluate compared to its forward direction; and the exact inverses of planar flow and residual flow are simply intractable.

Because of this polarity, in practice, we need to carefully choose the direction at which to apply these transforms. For example, if autoregressive flow is applied as a mapping from  $\mathbf{x}$  to  $\mathbf{z}$ , then we know that computing the density for given sample  $\mathbf{x}$  is efficient but the sampling of  $\mathbf{x}$  can be slow. Conversely, if it is applied by mapping  $\mathbf{z}$  to  $\mathbf{x}$ , then sampling from the target distribution (as well as obtaining density for such samples) can be executed efficiently, but obtaining the density of a given sample  $\mathbf{x}$  is a slow sequential process. In essence, which direction to apply these transforms depends on the application of interests, as detailed in Section 2.2.

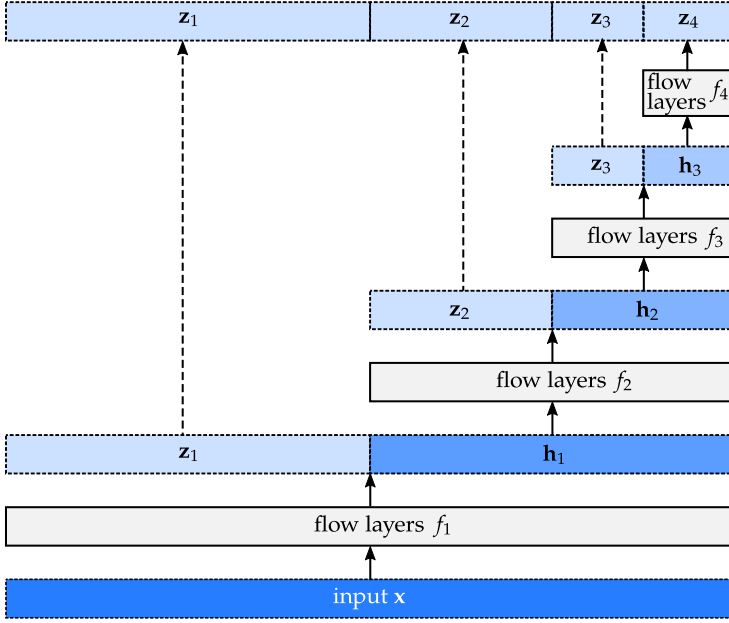
**Table 2.7:** Comparison of four types of flow layers covered in this section in terms of the tractability of their determinant Jacobian, and forward and inverse complexity. Intractable ones are marked as gray.

| Type of flow        | Jacobian  | forward  | inverse  |
|---------------------|---|--|--|
| Affine coupling     | <br>lower triangular<br>$\log \det J = \sum \log \text{diag}(J)$   | Forward and inverse share the same form.<br>Both are tractable and easy to evaluate.   |  |
| Autoregressive flow | <br>lower triangular<br>$\log \det J = \sum \log \text{diag}(J)$   | Single forward pass of $\text{AutoregressiveNN}_\theta$  | $N$ sequential evaluations of $\text{AutoregressiveNN}_\theta$<br>(much slower than forward) |
| Planar flow         | <br>$\mathbf{I} + \mathbf{A}$ , where $\text{rank}(\mathbf{A}) = 1$<br>matrix det. lemma:<br>$\det(\mathbf{I} + \mathbf{u}\mathbf{w}^T) = 1 + \mathbf{u}^T \mathbf{w}$ | Need to evaluate a linear layer with a single output activation.<br>(limited expressiveness due to the single-neuron bottleneck) | Intractable.<br>Can be approximated with some numerical iteration.                           |
| Residual flow       | <br>$\mathbf{I} + \mathbf{A}$ , where $\ \mathbf{A}\ _2 < 1$<br>$\log \det J = \sum_{k=1}^{\infty} (-1)^{k+1} \text{tr}(\mathbf{A}^k) / k$                             | Typical residual layer.<br>residual module must have a Lipschitz constant less than 1.   | Intractable.<br>Can be approximated with a fixed-point iteration.                            |

## 2.4 Network architectures

In this section, we introduce engineering practises of building normalizing flow networks. Out of the four basic building blocks introduced in the last section, we limit our focus to affine coupling layer, which is arguably the most widely adopted. Autoregressive flow is itself an important topic, and often regarded as a separate class of generative modeling. For that reason we treat it separately in Chapter 3.





**Figure 2.10:** Multi-scale architecture. Block with dotted edge represents tensor and with solid edge represent flow layers. Dashed line represent skip connection (identity transform). The split of output after each block of flow layers is also often referred to as *factor out* operation.

### 2.4.1 Multi-scale architecture

Individual flow layer tends to have limited capacity due to the invertibility constraint, and thus to achieve adequate expressive power we need to rely on building deeper networks. For this reason it is not uncommon to have normalizing flow networks that are more than 100 layers deep.

One property of invertible layers is that input and output dimension stay the same. Therefore, if the invertible layers are naively stacked on top of one another, the size of the activation tensor, no matter how deep it is, necessarily remains the same as the original input after each layer, as there is no dimension reduction from any flow layers. This, coupled with the fact that normalizing flows are normally deep, leads to a network design that can be very heavy in both memory and compute.

To limit the network size, a *multi-scale architecture* is proposed by [10] and soon gets adopted by many mature flow network designs [11, 14], where half of the activation tensor of certain intermediate layers is factored out as direct output without future processing, where the rest goes through additional flow layers. Specifically, the network is divided into  $L$  blocks of concatenated flow layers, where the output of flow block  $f_i$  is factored into two equal size tensors  $\mathbf{z}_i$  and  $\mathbf{h}_i$ .  $\mathbf{z}_i$  is directly treated as part of output and  $\mathbf{h}_i$  serves as input to downstream blocks. Figure 2.10 illustrates the case of  $L = 4$ .

$$\begin{aligned}
 (\mathbf{z}_i, \mathbf{h}_i) &= f_i(\mathbf{h}_{i-1}), \text{ for } i = 1, 2, \dots, L-1 \text{ and} \\
 \mathbf{z}_L &= f_L(\mathbf{h}_{L-1}), \text{ where} \\
 \mathbf{h}_0 &\triangleq \mathbf{x} \\
 \mathbf{z} &\triangleq \text{concat}(\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_L).
 \end{aligned}$$

The successive halving of activation sizes allows the complexity of the flow network to scale well with increased depth, and enables building deep network with good expressiveness. It also brings two additional benefits [10]:

[10]: Dinh et al. (2017), *Density estimation using Real NVP*

### multiple levels of representation

Since the final output comprises of segments that go through different number of layers, they can form different levels of representation. Intuitively speaking, high-frequency local details in the original input  $x$  tend to have smaller degree of correlation across different input elements. Hence it is easier for them to be transformed and decorrelated into independent components through a smaller number of flow layers compared to the low-frequency counterpart. One would expect  $z_1$  and  $z_L$  to capture the finest and coarsest scale information, respectively.

For image application, there is empirical evidence that this multi-scale architecture, when coupled with a prior that is auto-regressive across different scales  $z_1, z_2, \dots, z_L$ , achieves the decomposition of different levels of details [21].

### distributing loss at different layers of the network

One practical benefit of collecting outputs at various level of the network is that it allows gradient from the loss term to be distributed across intermediate layers, providing guidance at different depth levels of the network.

## 2.4.2 Other applications of invertible networks

The need for invertibility also arises in areas other than generative modeling.

One advantage of using invertible architecture is that it allows one to trade off computation complexity for reduced memory consumption during training. Specifically, back-propagation can be done without storing activations in the intermediate layers, as they can be recomputed in the reverse order given the final output<sup>11</sup> [22, 23]. This can become advantageous when there is need to compute gradient in a memory-constrained device.

Another desirable property of invertible architecture is that it preserves information of the input. It can be helpful in analyzing learned representation in the context of adversarial robustness [24].

## 2.4.3 Glow: a case study

In this subsection, we take a look at Glow[11], a well-known multi-scale normalizing flow architecture for the generative modeling of images, which uses affine coupling as its basic nonlinear invertible transform.

As introduced in Section 2.3.1, in an affine coupling layer, half of the input dimensions remain intact and get passed directly as part of output, with the other half going through an affine transform. The parameters of the affine transform are obtained as a learnable function of the first half of input. Since the first half remains unchanged, the affine transform parameters can be obtained in both directions, which then allows the layer to be easily inverted.

Since half of the input dimensions remain unchanged for a single layer, to device meaningful invertible network we necessarily need to permute

[11]: Often a non-invertible head is needed on top of the invertible backbone to obtain output in a certain dimension. In such a case, the last activation in the invertible segment of the network still needs to be stored.

[22]: Gomez et al. (2017), *The Reversible Residual Network: Backpropagation Without Storing Activations*

[23]: Chang et al. (2018), *Reversible Architectures for Arbitrarily Deep Residual Neural Networks*

[24]: Jacobsen et al. (2019), *Excessive Invariance Causes Adversarial Vulnerability*

[11]: Kingma et al. (2018), *Glow: Generative Flow with Invertible 1x1 Convolutions*

or shuffle the activation in between affine coupling layers. There are two design aspects: (1) how to divide tensor into two halves (2) how to shuffle the activation dimensions in between successive affine coupling layers.

In the design of NICE [9], the input image is flattened as a 1-dimensional array. The two halves of the input correspond to even and odd components in the 1-d array, and are alternated in successive affine coupling layers, i.e., the part that remains unchanged in one layer is transformed in the next. In RealNVP [10], the multiple-scale architecture is proposed and the input is either divided along the channel-dimension or spatial-dimension following a checkerboard pattern. Specifically, in each scale, the input first goes through three affine coupling layers with alternating checkerboard masks, followed by a space-to-depth operator that packs spatial dimension into channel dimension, and then three affine coupling layers with alternating channel-wise masking.

Glow [11] further generalizes the fixed shuffling scheme in NICE [9] and RealNVP [10] to a learnable invertible 1x1 convolution. It inherited the multi-scale architecture used in RealNVP, and in each scale, the input first goes through an space-to-depth (a.k.a. squeeze) operation, followed by  $K$  flow steps, where each step is composed of a 1x1 convolution layer, which is invertible whenever the parameter matrix in the 1x1 convolution is full-rank, and an affine coupling layer with the input divided in channel dimension. For details please see Figure 2.11 and the pseudo code right below.

[9]: Dinh et al. (2015), 'NICE: Non-linear Independent Components Estimation'

[10]: Dinh et al. (2017), *Density estimation using Real NVP*

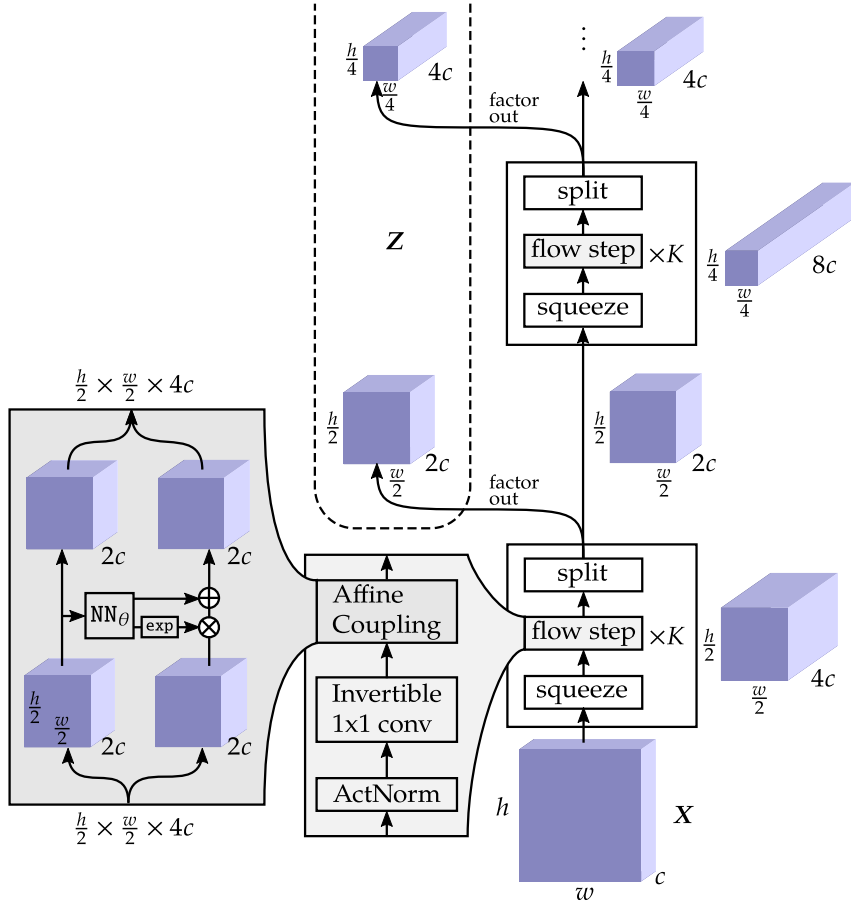


Figure 2.11: Multi-scale architecture with squeeze and factor out (split).

```
import torch
from torch import nn
```

```
class ActNorm(nn.Module):
```

```
def __init__(self, ch):
    super().__init__()
    self.t = nn.Parameter(
        torch.zeros(1, ch))
    self.s = nn.Parameter(
        torch.ones(1, ch))
```

```
def logdet(self, x):
    _, _, h, w = x.shape
    one_pix = self.s.abs().log().sum()
    return h * w * one_pix
```

```
def forward(self, x, reverse=False):
    if reverse:
        output = x / self.s - self.t
        return output, -self.logdet(x)
    else:
        output = self.s * (x + self.t)
        return output, self.logdet(x)
```

```
class Invert1Conv(nn.Module):
```

```
def __init__(self, ch):
    super().__init__()
    w = torch.zeros((ch, ch))
    w = nn.init.orthogonal_(w)
    self.w = nn.Parameter(w)
```

```
def logdet(self, x):
    _, _, h, w = x.shape
    one_pix = \
        torch.linalg.slogdet(self.w)[1]
    return h * w * one_pix
```

```
def forward(self, x, reverse=False):
    w = self.w
    if reverse: w = w.inverse()
    output = nn.functional.\
        conv2d(x, w[...], None, None)
    if reverse:
        return output, self.logdet(x)
    else:
        return output, - self.logdet(x)
```

```
class AffineCoupling(nn.Module):
```

```
def __init__(self, ch, hid=512):
    super().__init__()
    self.nn = nn.Sequential(
        nn.Conv2d(
            ch // 2, hid, 3, padding=1),
        nn.ReLU(),
        nn.Conv2d(hid, hid, 1),
        nn.Conv2d(hid, ch, 3, padding=1))
```

```
def forward(self, x, reverse=False):
    x_a, x_b = x.chunk(chunks=2, dim=1)
    log_s, t = self.nn(x_a).chunk(
        chunks=2, dim=1)
    s = torch.exp(log_s)
    if reverse:
        output_b = (x_b - t) / s
        logdet = - log_s.sum()
    else:
        output_b = x_b * s + t
        logdet = log_s.sum()
    output = torch.cat(
        [x_a, output_b], dim=1)
    return output, logdet
```

## 2.5 Continuous normalizing flows\*

So far in our discussion the flow networks are built by stacking a discrete number of invertible layers. It turns out that for the case of residual flow layers, there is a way to generalize the concept of discrete layers into a continuous one, which we introduce in this section.

### 2.5.1 Neural ODE

Neural ODE [25] is built on the insight that network with a sequence of residual layers can be thought of as Euler method in solving an ordinary differential equation (ODE).

$$\left. \begin{array}{l} \mathbf{z}_1 = \mathbf{z}_0 + \text{NN}_{\theta_0}(\mathbf{z}) \\ \mathbf{z}_2 = \mathbf{z}_1 + \text{NN}_{\theta_1}(\mathbf{z}_1) \\ \dots \end{array} \right\} \longrightarrow \frac{\partial \mathbf{z}_t}{\partial t} = \text{NN}_{\theta_t}(\mathbf{z}_t)$$

So instead of specifying a residual network with a discrete sequence of hidden layers  $\text{NN}_{\theta_n}$  and hidden state  $\mathbf{z}_n$ , Neural ODE defines a continuous-depth network by parametrizing the derivative of the hidden state  $\mathbf{z}_t$  with a neural network, denoted as  $r_{\theta}(\mathbf{z}, t)$ ,

$$\frac{\partial \mathbf{z}_t}{\partial t} = r_{\theta}(\mathbf{z}_t, t). \quad (2.9)$$

The input to the network is  $\mathbf{z}_0$  and the output is  $\mathbf{z}_T$  for a predetermined  $T$ .

*Picard–Lindelöf theorem* states that if  $r_{\theta}(\mathbf{z}, t)$  is Lipschitz continuous in  $\mathbf{z}$  and continuous in  $t$ , then for any given time  $t_0$  and any initial value  $\mathbf{z}_{t_0}$  – referred to as an *initial value* – the solution to the above ODE is unique. Uniqueness of the solution given an initial value implies that the network is invertible, as it suggests that for a given  $\mathbf{z}_0$ ,  $\mathbf{z}_T$  is unique and vice versa.

Note that different from residual flow described in Section 2.3.4 which requires Lipschitz constant of the residual module to be less than 1 for invertibility, a Neural ODE is invertible so long as  $r_{\theta}$  is Lipschitz continuous in  $\mathbf{z}$ , without imposing any constraint on the exact value of the Lipschitz constant. This can be seen by expressing  $\mathbf{z}_t$  as its first order Taylor expansion

$$\begin{aligned} \mathbf{z}_{t+\epsilon} &= \mathbf{z}_t + \epsilon \frac{\partial \mathbf{z}_t}{\partial t} + o(\epsilon) \\ &= \mathbf{z}_t + \epsilon r_{\theta}(\mathbf{z}_t, t) + o(\epsilon), \end{aligned} \quad (2.10)$$

which has the same form as the residual flow in Equation 2.7. If we ignore the second order term, then from Proposition 2.3.1 we know that the mapping is invertible as long as  $\epsilon r_{\theta}$  has Lipschitz constant smaller than 1. Since  $\epsilon$  can be arbitrarily small, we just need  $r_{\theta}$  to be Lipschitz continuous to guarantee invertibility.

[25]: Chen et al. (2018), *Neural Ordinary Differential Equations*

Lipschitz continuous is a stronger form of uniform continuous.

Uniform continuity:  
 $\forall \epsilon, \exists \delta, \text{ s.t. } \forall \mathbf{x}, \mathbf{y} \text{ with } \|\mathbf{x} - \mathbf{y}\| < \delta, \text{ we have } \|f(\mathbf{x}) - f(\mathbf{y})\| < \epsilon.$

Lipschitz continuity:  
 $\exists K, \text{ s.t. } \forall \mathbf{x}, \mathbf{y}, \text{ we have } \|f(\mathbf{x}) - f(\mathbf{y})\| < K\|\mathbf{x} - \mathbf{y}\|$

## 2.5.2 Change of variables formula for ODE

By treating this first order Taylor expansion in Equation 2.10 as a discretized step of the ODE, and applying the change of variable formula in Proposition 2.1.1, we obtain

$$\begin{aligned}
p(\mathbf{z}_{t+\epsilon}) &= p(\mathbf{z}_t) \left| \det \frac{\partial \mathbf{z}_{t+\epsilon}}{\partial \mathbf{z}_t} \right| \\
\log p(\mathbf{z}_{t+\epsilon}) &= \log p(\mathbf{z}_t) - \log \left| \det \frac{\partial \mathbf{z}_{t+\epsilon}}{\partial \mathbf{z}_t} \right| \\
&\stackrel{(a)}{=} \log p(\mathbf{z}_t) - \log \det \left( \mathbf{I} + \epsilon \frac{\partial r_\theta}{\partial \mathbf{z}_t} + o(\epsilon) \right) \\
&\stackrel{(b)}{=} \log p(\mathbf{z}_t) - \text{tr} \log \left( \mathbf{I} + \epsilon \frac{\partial r_\theta}{\partial \mathbf{z}_t} + o(\epsilon) \right) \\
&\stackrel{(c)}{=} \log p(\mathbf{z}_t) - \text{tr} \left( \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\left( \epsilon \frac{\partial r_\theta}{\partial \mathbf{z}_t} + o(\epsilon) \right)^k}{k} \right) \\
&\stackrel{(d)}{=} \log p(\mathbf{z}_t) - \epsilon \text{tr} \left( \frac{\partial r_\theta}{\partial \mathbf{z}_t} \right) + o(\epsilon).
\end{aligned}$$

Step (a) follows from Equation 2.10. We remove the absolute sign assuming that  $\epsilon$  is small enough such that the determinant is positive; step (b) applies matrix identity of  $\log \det \mathbf{A} = \text{tr} \log \mathbf{A}$  (which has a scalar analog of  $\log \Pi = \sum \log$ ); step (c) is the definition of matrix logarithm; and in step (d) the higher order terms are collapsed in  $o(\epsilon)$ .

By dividing both side of the equation by  $\epsilon$  and take the limit of  $\epsilon \rightarrow 0$ , we obtain below the continuous version of the change of variable equation,

$$\begin{aligned}
\frac{\log p(\mathbf{z}_{t+\epsilon}) - \log p(\mathbf{z}_t)}{\epsilon} &= -\text{tr} \left( \frac{\partial r_\theta}{\partial \mathbf{z}_t} \right) + \frac{o(\epsilon)}{\epsilon} \\
\Rightarrow \frac{\partial \log p(\mathbf{z}_t)}{\partial t} &= -\text{tr} \left( \frac{\partial r_\theta}{\partial \mathbf{z}_t} \right),
\end{aligned}$$

which is formally stated in the proposition below

**Proposition 2.5.1** (Instantaneous change of variable formula) *For an ODE  $\frac{\partial \mathbf{z}_t}{\partial t} = r_\theta(\mathbf{z}_t, t)$ , where  $r_\theta(\mathbf{z}, t)$  is Lipschitz continuous in  $\mathbf{z}$  and continuous in  $t$ , the log density of  $\mathbf{z}_t$  follows the following ODE*

$$\frac{\partial \log p(\mathbf{z}_t)}{\partial t} = -\text{tr} \left( \frac{\partial r_\theta}{\partial \mathbf{z}_t} \right)$$

By combining the above change of variable formula with Equation 2.9, we can obtain an ODE with extended state space that captures the evolution of both the hidden state  $\mathbf{z}_t$  itself and its log density  $\log p(\mathbf{z}_t)$ .

$$\frac{\partial}{\partial t} \begin{bmatrix} \mathbf{z}_t \\ \log p(\mathbf{z}_t) \end{bmatrix} = \begin{bmatrix} r_\theta(\mathbf{z}_t, t) \\ -\text{tr}(\partial r_\theta / \partial \mathbf{z}_t) \end{bmatrix}.$$

An interesting contrast between discrete change of variable formula (Proposition 2.1.1) and its instantaneous counterpart (Proposition 2.5.1)

$$\log \det(\mathbf{I} + \mathbf{A}) = \text{tr} \log(\mathbf{I} + \mathbf{A})$$

$$\begin{aligned}
&= \text{tr} \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\mathbf{A}^k}{k} \\
&= \sum_{k=1}^{\infty} (-1)^{k+1} \frac{\text{tr}(\mathbf{A}^k)}{k}
\end{aligned}$$

is that the former requires computation of the *determinant* of the Jacobian matrix of the neural network that captures the transform, while the latter requires computation of the *trace* of the Jacobian matrix of the neural network that captures the derive of the hidden state (i.e.,  $r_\theta$ ). Thus, we should favor the design of  $r_\theta$  whose trace of Jacobian is easy to compute, preferably in closed-form.

In [25], the authors propose  $r_\theta(\mathbf{z}, t) = \mathbf{u}h(\mathbf{w}^T\mathbf{z} + b)$ , which can be viewed as a continuous form of the planar flow defined in Equation 2.6. The trace of its Jacobian has a closed form below.

$$\begin{aligned} \text{tr}\left(\frac{\partial r_\theta}{\partial \mathbf{z}}\right) &= \text{tr}\left(\mathbf{u}\frac{\partial h(\mathbf{w}^T\mathbf{z} + b)}{\partial(\mathbf{w}^T\mathbf{z} + b)}\frac{\partial \mathbf{w}^T\mathbf{z} + b}{\partial \mathbf{z}}\right) = \\ &= \text{tr}\left(h'(\mathbf{w}^T\mathbf{u} + b)\mathbf{u}\mathbf{w}^T\right) \\ &= h'(\mathbf{w}^T\mathbf{u} + b)\mathbf{w}^T\mathbf{u} \end{aligned}$$

trace operation is invariant under cyclic permutations

$$\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$$

It is worth noting that in the continuous version, we do not need to enforce additional condition for invertibility – as long as  $h()$  is Lipschitz continuous, as is the case for most of the non-linear activation functions, invertibility is guaranteed. Similarly, one can generalize Sylvester flow defined in Section 2.3.3 to its continuous ODE version with closed-form trace Jacobian, which we omit here.

## 2.6 Summary

In this chapter we introduced normalizing flows based model for the learning of continuous multi-dimensional distributions. The modeled distribution is obtained by transforming a simple base distribution through a learned non-linear invertible transform. The density of the modeled distribution is linked to that of the base distribution through the change of variable formula. The log-determinant-Jacobian term in the formula accounts for the rate of dilution or concentration of the densities incurred by the transform.

Two design constraints of normalizing flow networks are: (1) they have to be invertible (2) their determinant Jacobians should be easy to compute. Early literature focus on introducing various types of base layers that trade off flexibility and tractability [8–10, 13, 14, 26]. Mature network designs often adopt coupling layer when bidirectional tractability is needed [11, 27], and autoregressive flow when only we only need to evaluate a single direction [7, 28]. These two cases correspond to the two applications below:

- (a) direct modeling of real world data distribution
- (b) modeling of approximation posterior in the context of variation autoencoder

These two are quite different in their tractability requirement of the network. For the former, we typically need to evaluate the network in both directions: from  $\mathbf{X}$  to  $\mathbf{Z}$  to compute likelihood score during training, and from  $\mathbf{Z}$  to  $\mathbf{X}$  when sampling is needed at inference time. Well-known solutions in this space [11, 27, 29] follow the recipe of tractable coupling layer with learned shuffling, the use of very deep networks to achieve

desired expressiveness, and the multi-scale (factor-out) structure to control network complexity.

In contrast, for the latter we only need to execute in the direction of base distribution to the modeled one, which allows us to use transforms that are unbalanced in the tractability/speed of the two directions. A popular choice is autoregressive flow, as proposed in [7], which we will discuss in the VAE chapter.

Normalizing flow based generative models come with two notable limitations: (1) There is no dimension reduction. The invertibility constraint requires the input dimension to be exactly the same as the output dimension. (2) They are not immediately applicable for the learning of *discrete* distributions. For discrete distributions, the change of variable formula degenerates to a direct one-to-one mapping of probabilities mass without the log-det-Jacobian term. Any invertible mapping only shuffles the discrete probability masses defined in the base distribution, which limits its expressiveness. There are works that extends normalizing flow beyond these two limitations: notably in [30] the authors propose methods to jointly learn the low-dimensional manifold of the data as well as the density on the manifold. A discrete version of normalizing flow is proposed in [21] (and refined in [31]) where the base distribution is obtained from a learnable block autoregressive model, with an additive coupling layer with integer operation.

[7]: Kingma et al. (2017), *Improving Variational Inference with Inverse Autoregressive Flow*

[30]: Brehmer et al. (2020), *Flows for simultaneous manifold learning and density estimation*

[21]: Hoogeboom et al. (2019), *Integer Discrete Flows and Lossless Compression*

## Food for thought

**Question 1** Can BatchNorm or LayerNorm layers be used in invertible networks? Can Dropout be applied?

**Question 2** In a network where multiple affine coupling layers are used with alternating pattern of input (e.g., [9]), how many layers are needed to allow all dimensions to influence one another?

**Question 3** Is a function invertible if its Jacobian is invertible everywhere?

**Question 4** Why do we need to multiply the log determinant Jacobian of the 1x1 convolution by  $h \times w$  in the pseudo code below Figure 2.11?

**Question 5** Is there a one-to-one mapping from  $\mathbb{R}^2$  to  $\mathbb{R}^1$  (or more generally from  $\mathbb{R}^m$  to  $\mathbb{R}^n$  where  $m \neq n$ )? If so, can we use it for normalizing flows?

**Question 6** What is the relationship between the Lipschitz constant of a function and its Jacobian?



# Bibliography

Here are the references in citation order.

- [1] Ian J. Goodfellow. *On Distinguishability Criteria for Estimating Generative Models*. 2015 (cited on page 1).
- [2] Ferenc Huszár. *How (not) to Train your Generative Model: Scheduled Sampling, Likelihood, Adversary?* 2016 (cited on page 2).
- [3] Shane Barratt and Rishi Sharma. *A Note on the Inception Score*. 2018 (cited on page 2).
- [4] Martin Heusel et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2017 (cited on page 2).
- [5] Mehdi S. M. Sajjadi et al. *Assessing Generative Models via Precision and Recall*. 2018 (cited on page 2).
- [6] Tuomas Kynkäänniemi et al. *Improved Precision and Recall Metric for Assessing Generative Models*. 2019 (cited on page 2).
- [7] Diederik P. Kingma et al. *Improving Variational Inference with Inverse Autoregressive Flow*. 2017 (cited on pages 15, 21, 33–35).
- [8] Danilo Jimenez Rezende and Shakir Mohamed. ‘Variational Inference with Normalizing Flows’. In: (2016) (cited on pages 15, 21, 22, 33).
- [9] Laurent Dinh, David Krueger, and Yoshua Bengio. ‘NICE: Non-linear Independent Components Estimation’. In: (2015) (cited on pages 18, 19, 29, 33, 34).
- [10] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. *Density estimation using Real NVP*. 2017 (cited on pages 19, 27, 29, 33).
- [11] Diederik P. Kingma and Prafulla Dhariwal. *Glow: Generative Flow with Invertible 1x1 Convolutions*. 2018 (cited on pages 19, 27–29, 33).
- [12] George Papamakarios, Theo Pavlakou, and Iain Murray. *Masked Autoregressive Flow for Density Estimation*. 2018 (cited on page 21).
- [13] Rianne van den Berg et al. *Sylvester Normalizing Flows for Variational Inference*. 2019 (cited on pages 22, 23, 33).
- [14] Jens Behrmann et al. *Invertible Residual Networks*. 2019 (cited on pages 23, 24, 27, 33).
- [15] Ricky T. Q. Chen et al. *Residual Flows for Invertible Generative Modeling*. 2019 (cited on pages 23, 24).
- [16] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2016 (cited on page 23).
- [17] Bolin Gao and Lacra Pavel. *On the Properties of the Softmax Function with Application in Game Theory and Reinforcement Learning*. 2018 (cited on page 25).
- [18] Henry Gouk et al. *Regularisation of Neural Networks by Enforcing Lipschitz Continuity*. 2018 (cited on page 25).
- [19] Takeru Miyato et al. *Spectral normalization for generative adversarial networks*. 2018 (cited on page 25).
- [20] Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. *Lipschitz-Margin Training: Scalable Certification of Perturbation Invariance for Deep Neural Networks*. 2018 (cited on page 25).
- [21] Emiel Hoogeboom et al. *Integer Discrete Flows and Lossless Compression*. 2019 (cited on pages 28, 34).
- [22] Aidan N. Gomez et al. *The Reversible Residual Network: Backpropagation Without Storing Activations*. 2017 (cited on page 28).
- [23] Bo Chang et al. *Reversible Architectures for Arbitrarily Deep Residual Neural Networks*. 2018 (cited on page 28).
- [24] Jörn-Henrik Jacobsen et al. *Excessive Invariance Causes Adversarial Vulnerability*. 2019 (cited on page 28).

- [25] Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2018 (cited on pages 31, 33).
- [26] Will Grathwohl et al. *FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models*. 2019 (cited on page 33).
- [27] Jonathan Ho et al. *Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design*. 2019. URL: <http://arxiv.org/abs/1902.00275> (cited on page 33).
- [28] Arash Vahdat and Jan Kautz. *NVAE: A Deep Hierarchical Variational Autoencoder*. Ed. by H. Larochelle et al. 2020. URL: <https://proceedings.neurips.cc/paper/2020/file/e3b21256183cf7c2c7a66be163579d37-Paper.pdf> (cited on page 33).
- [29] Ryan Prenger, Rafael Valle, and Bryan Catanzaro. *Waveglow: A Flow-based Generative Network for Speech Synthesis*. 2019. DOI: [10.1109/ICASSP.2019.8683143](https://doi.org/10.1109/ICASSP.2019.8683143) (cited on page 33).
- [30] Johann Brehmer and Kyle Cranmer. *Flows for simultaneous manifold learning and density estimation*. Ed. by H. Larochelle et al. 2020. URL: <https://proceedings.neurips.cc/paper/2020/file/051928341be67dcb03f0e04104d9047-Paper.pdf> (cited on page 34).
- [31] Rianne van den Berg et al. *{IDF}++: Analyzing and Improving Integer Discrete Flows for Lossless Compression*. 2021. URL: <https://openreview.net/forum?id=MB0yiNnYthd> (cited on page 34).
- [32] XuanLong Nguyen, Martin J Wainwright, and Michael I Jordan. 'Estimating divergence functionals and the likelihood ratio by convex risk minimization'. In: *IEEE Trans. Inf. Theory* 56.11 (2010), pp. 5847–5861 (cited on page 44).
- [33] Ian Goodfellow et al. 'Generative adversarial nets'. In: *Adv. Neural Inf. Proc. Syst.* 2014, pp. 2672–2680 (cited on page 45).
- [34] P. Vincent. 'A Connection Between Score Matching and Denoising Autoencoders'. In: *Neural Comput.* 23.7 (2011), pp. 1661–1674. DOI: [10.1162/NECO\\_a\\_00142](https://doi.org/10.1162/NECO_a_00142) (cited on page 59).