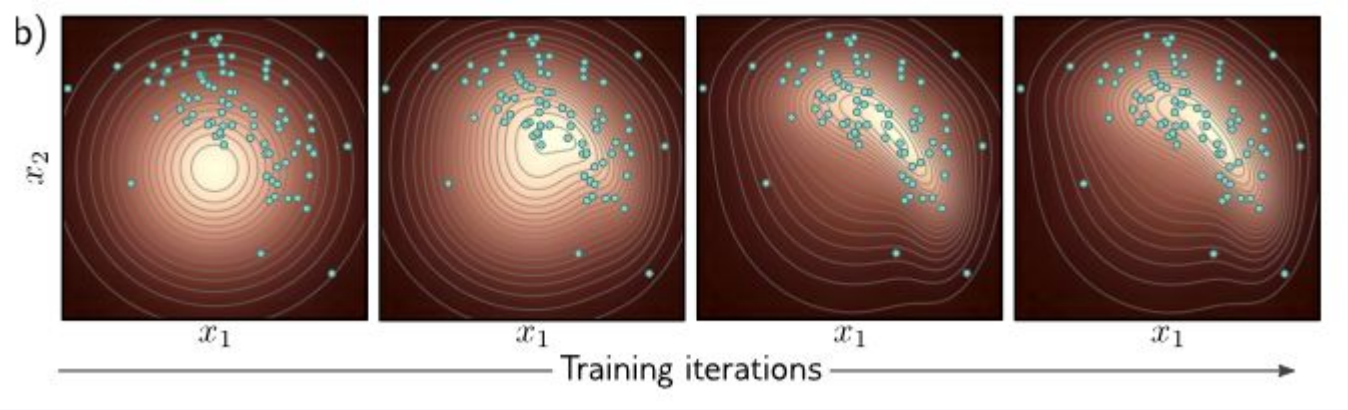
Normalizing Flow Models

Vijay Raghavan

Probabilistic models



- Probabilistic models learn a probability distribution over the training data.
- As training proceeds (left to right), the likelihood of the real examples increases under this distribution
- which can be used to draw new samples and assess the probability of new data points.

Prince, S. J. (2023). *Understanding Deep Learning*. MIT press

Normalizing flows

- These learn a probability model by transforming a simple distribution into a more complicated one using a deep network.
- Normalizing flows can both sample from this distribution and evaluate the probability of new examples.
- However, they require specialized architecture: each layer must be invertible.
- In other words, it must be able to transform data in both directions

Prince, S. J. (2023). *Understanding Deep Learning*. MIT press

Normalizing flows, VAEs and Autoregressive models

- Share similarities with both autoregressive models and variational autoencoders.
- Like autoregressive models, normalizing flows are able to explicitly and tractably model the data-generating distribution .
- Like VAEs, normalizing flows attempt to map the data into a simpler distribution, such as a Gaussian distribution.
- The key difference is that normalizing flows place a constraint on the form of the mapping function
- That it is invertible and can therefore be used to generate new data points

Prince, S. J. (2023). *Understanding Deep Learning*. MIT press

Invertible/Bijective Transformations

Invertible/Bijective Transformations

Maps each input element to a unique output element, and vice versa

Formal definition:

- Function $f : X \to Y$ is invertible if and only if:
  1. f is injective (one-to-one): for every $y \in Y$, there is at most one $x \in X$ such that $f(x) = y$
  2. f is surjective (onto): for every $y \in Y$, there is at least one $x \in X$ such that $f(x) = y$
- Inverse function $f^{-1} : Y \to X$ exists, such that:
  1. $f^{-1}(f(x)) = x$ for all $x \in X$
  2. $f(f^{-1}(y)) = y$ for all $y \in Y$

Properties and Applications in Normalizing Flows

- Properties of invertible transformations:
  - Preservation of information: no loss of information during transformation
  - Composition: composition of two invertible transformations is also invertible
  - Determinant of Jacobian: non-zero everywhere, crucial for normalizing flows
- Applications in normalizing flows:
  - Map complex data distribution to simpler base distribution (e.g., Gaussian) and vice versa
  - Ensures no information loss and enables density estimation and sampling

Examples of invertible transformations in normalizing flows

1. Affine transformations

   $$x \mapsto Ax + b$$

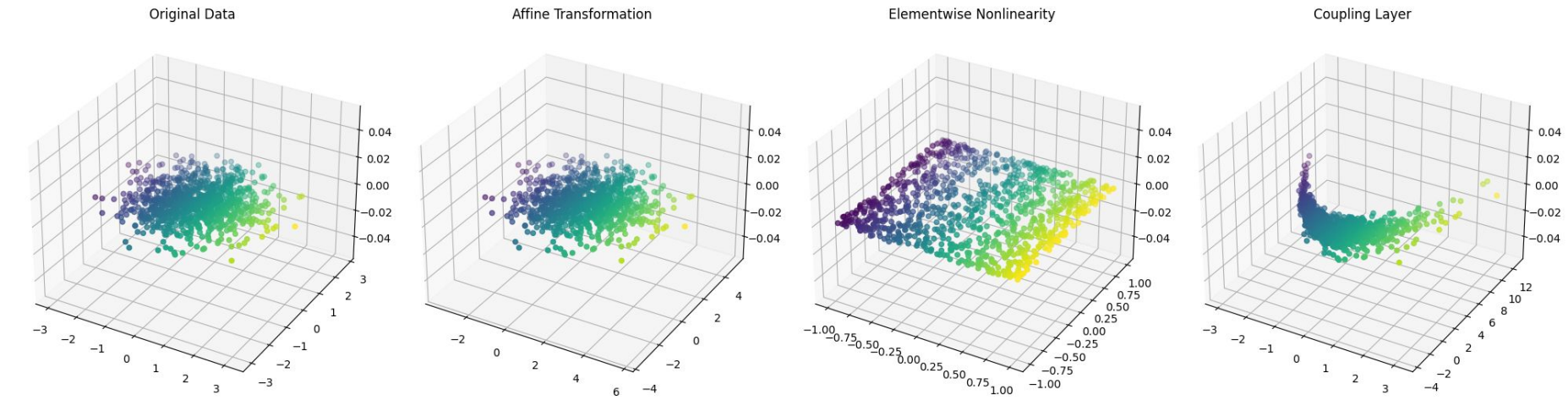
2. Elementwise nonlinearities

   $$x \mapsto f(x)$$

   Where f is monotonically increasing

3. Coupling layers:

   $$x = (x_1, x_2) \mapsto (x_1, x_2 \odot \exp(s(x_1)) + t(x_1))$$

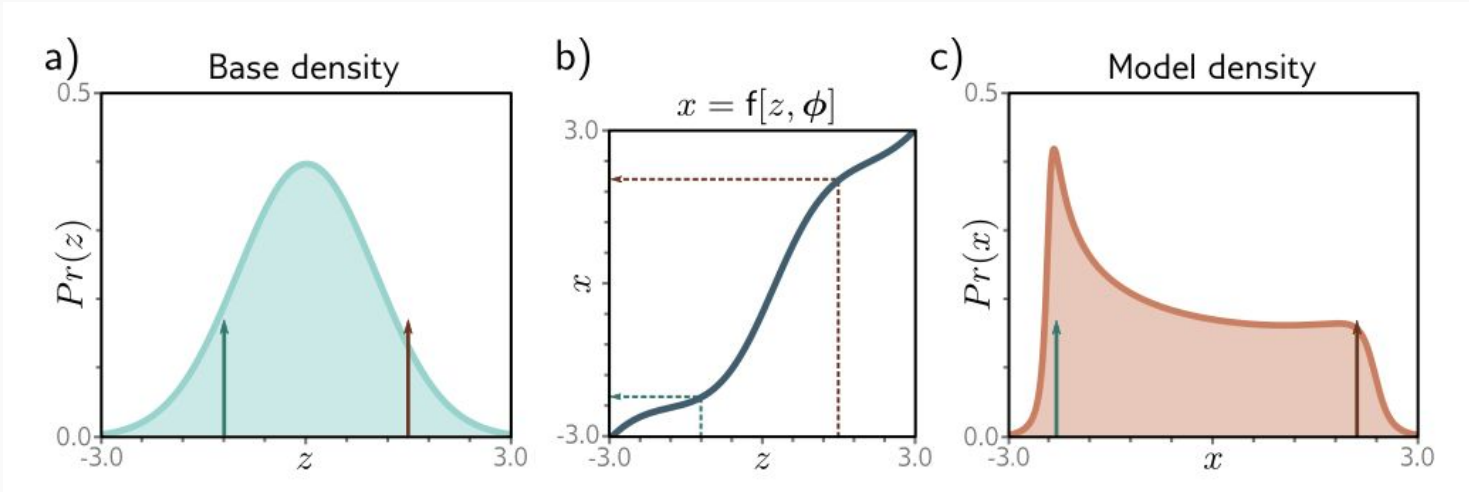

Original Data
Affine Transformation
Elementwise Nonlinearity
Coupling Layer

Probability Transformation

Measuring Probability

- Measuring the probability of a data point x is more challenging.
- Consider applying a function $f[z, \phi]$ to random variable z with known density $Pr(z)$.
- The probability density will decrease in areas that are stretched by the function and increase in areas that are compressed so that the area under the new distribution remains one.
- The degree to which a function $f[z,\phi]$ stretches or compresses its input depends on the magnitude of its gradient.
- If a small change to the input causes a larger change in the output, it stretches the function.
- If a small change to the input causes a smaller change in the output, it compresses the function

Transforming probability distributions



a) The base density is a standard normal defined on a latent variable *z*.

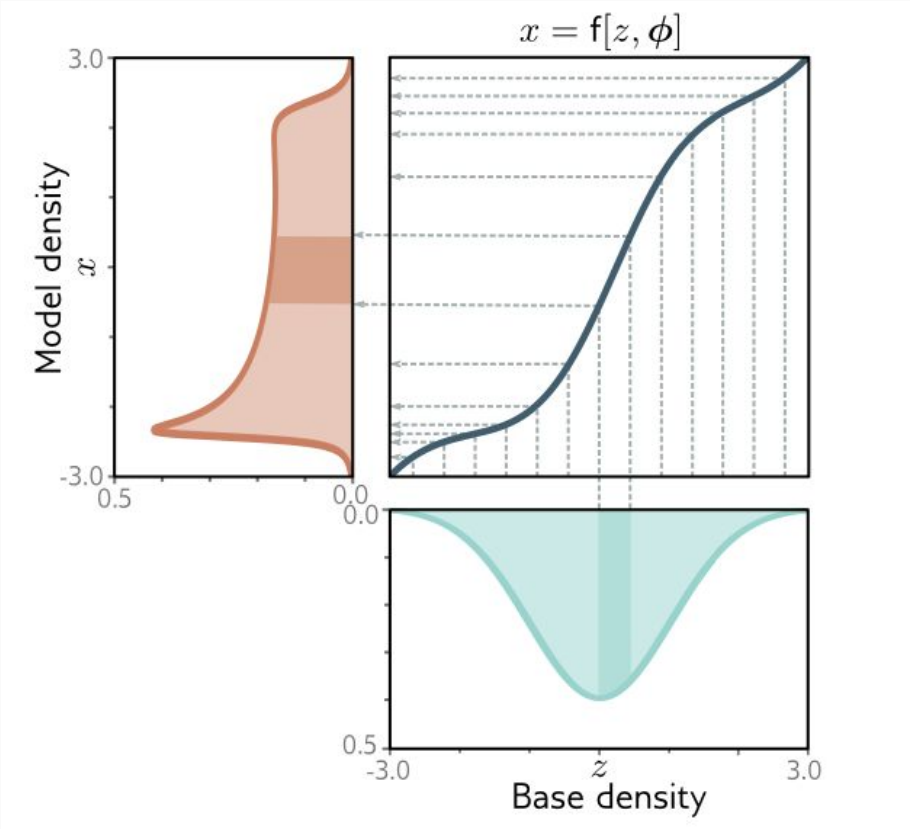
b) This variable is transformed by a function *x = f[z, φ]* to a new variable *x*, which has a new distribution.

c) To sample from this model, we draw values *z* from the base density (green and brown arrows in panel (a) show two examples).

We pass these through the function *f[z, φ]* as shown by dotted arrows in panel (b) to generate the values of *x*, which are indicated as arrows in panel *(c)*.

Prince, S. J. (2023). *Understanding Deep Learning*. MIT press

Transforming distributions

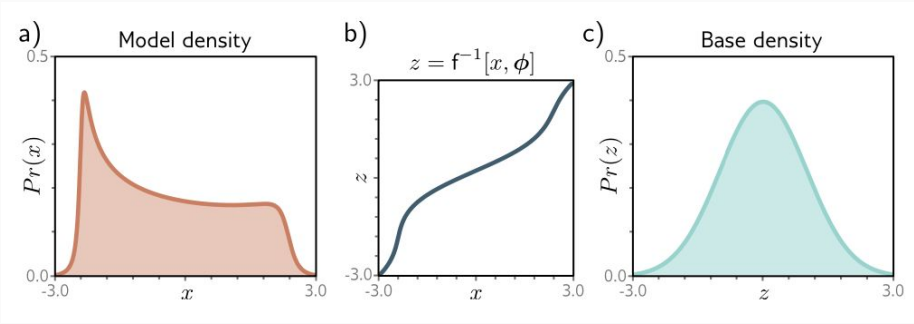


- The base density (cyan, bottom) passes through a function (blue curve, top right) to create the model density (orange, left).
- Consider dividing the base density into equal intervals (gray vertical lines).
- The probability mass between adjacent lines must remain the same after transformation.
- The cyan-shaded region passes through a part of the function where the gradient is larger than one, so this region is stretched.
- Consequently, the height of the orange-shaded region must be lower so that it retains the same area as the cyan-shaded region.
- In other places (e.g., z = −2), the gradient is less than one, and the model density increases relative to the base density

Prince, S. J. (2023). *Understanding Deep Learning*. MIT press

Inverse mapping (normalizing direction)

- If the function is invertible, then it's possible to transform the model density back to the original base density.
- The probability of a point x under the model density depends partly on the probability of the equivalent point z under the base density.
- More precisely, the probability of data x under the transformed distribution is:



$$Pr(x|\phi) = \left|\frac{\partial \mathrm{f}[z,\phi]}{\partial z}\right|^{-1} \cdot Pr(z),$$

Where, $z = \mathrm{f}^{-1}[x,\varphi]$ is the latent variable that created *x*.

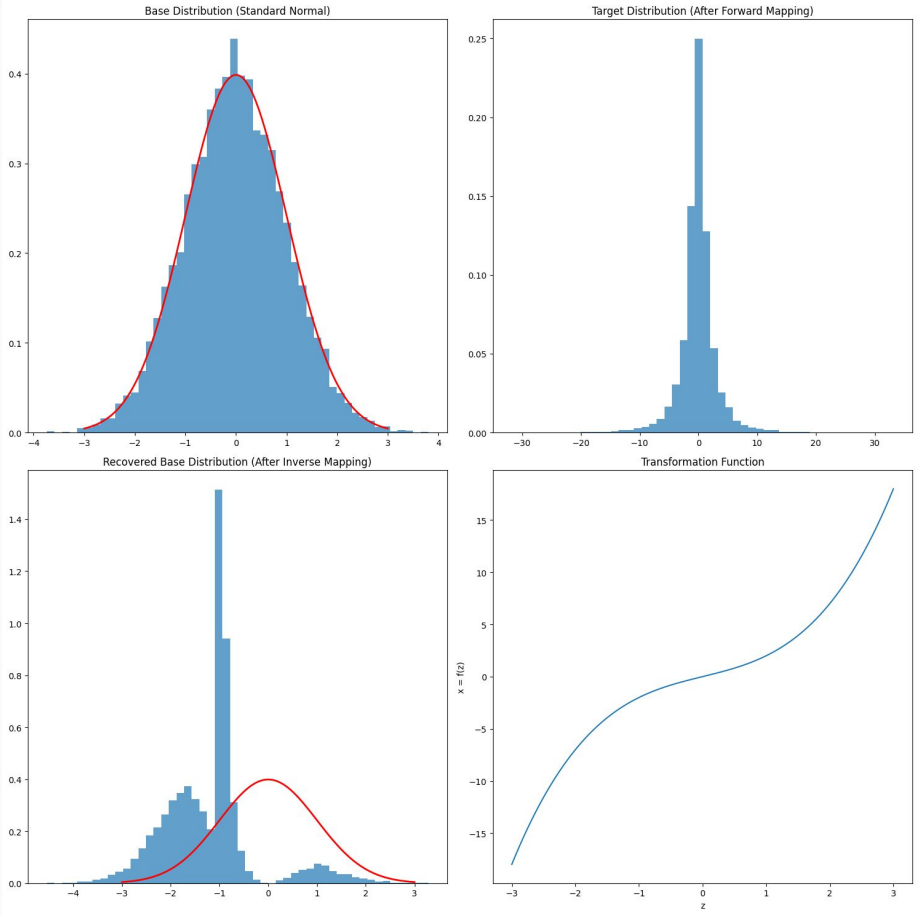
- The term *Pr(z)* is the original probability of this latent variable under the base density.
- This is moderated according to the magnitude of the derivative of the function.
- If this is greater than one, then the probability decreases. If it is smaller, the probability increases.

Prince, S. J. (2023). *Understanding Deep Learning*. MIT press

Forward and Inverse mappings

- To draw samples from the distribution, we need the forward mapping $x = f[z, \varphi]$, but to measure the likelihood, we need to compute the inverse $z = f^{-1}[x,\varphi]$.
- Hence, we need to choose $f[z, \varphi]$ judiciously so that it is invertible.
- The forward mapping is sometimes termed the generative direction.
- The base density is usually chosen to be a standard normal distribution.
- Hence, the inverse mapping is termed the normalizing direction since this takes the complex distribution over x and turns it into a normal distribution over z

Base and Target distributions - Issues



Learning

To learn the distribution, we find parameters φ that maximize the likelihood of the training data $\{x_i\}_{i=1}^{I}$ or equivalently minimize the negative log-likelihood in data that are independent and identically distributed:

$$\underset{\phi}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \log \left[\left| \frac{\partial \mathrm{f}[z_i, \boldsymbol{\phi}]}{\partial z_i} \right| \right] - \log \left[Pr(z_i) \right] \right]$$

Normalizing flows - General case

1. We are modeling a multivariate probability distribution $Pr(x)$ by transforming a simpler multivariate base distribution $Pr(z)$ using a function $x = f(z,\varphi)$ defined by a neural network with parameters φ.

2. Pr(z) is some simple base distribution that we can easily sample from. To generate a sample x from the distribution Pr(x), we:

   (a) Draw a sample $z \sim Pr(z)$

   (b) Compute $x = f(z,\varphi)$

3. The likelihood Pr(x|φ) of a sample x under this distribution is given by:

   $Pr(x|\varphi) = |\partial f/\partial z|^{-1} \cdot Pr(z)$

   Where:

   - $|\partial f/\partial z|$ is the absolute determinant of the Jacobian matrix. This measures how f changes volumes locally around the point z.
   - $Pr(z)$ is the density of z under the base distribution.

4. So by transforming a simple base density through a neural network, we can model a more complex distribution $Pr(x)$. The Jacobian term accounts for how the transformation f distorts volumes/densities.

Forward Mapping

1. The forward mapping f(z,φ) that transforms the base distribution z to the complex distribution x is defined as a composition of a series of neural network layers f_k:

$$\mathbf{x} = \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}] = \mathbf{f}_K\Big[\mathbf{f}_{K-1}\Big[\ldots \mathbf{f}_2[\mathbf{f}_1[\mathbf{z}, \boldsymbol{\phi}_1], \boldsymbol{\phi}_2], \ldots \boldsymbol{\phi}_{K-1}\Big], \boldsymbol{\phi}_K\Big]$$

2. The inverse mapping $f^{-1}(x,\varphi)$ that maps x back to z is defined as applying the inverse of each layer in reverse order:

$$\mathbf{z} = \mathbf{f}^{-1}[\mathbf{x}, \boldsymbol{\phi}] = \mathbf{f}_1^{-1}\Big[\mathbf{f}_2^{-1}\Big[\ldots \mathbf{f}_{K-1}^{-1}[\mathbf{f}_K^{-1}[\mathbf{x}, \boldsymbol{\phi}_K], \boldsymbol{\phi}_{K-1}], \ldots \boldsymbol{\phi}_2\Big], \boldsymbol{\phi}_1\Big]$$

3. The Jacobian ∂f/∂z for the forward mapping is computed by multiplying the Jacobians of each layer:

$$\frac{\partial \mathbf{f}[\mathbf{z}, \boldsymbol{\phi}]}{\partial \mathbf{z}} = \frac{\partial \mathbf{f}_K[\mathbf{f}_{K-1}, \boldsymbol{\phi}_K]}{\partial \mathbf{f}_{K-1}} \cdot \frac{\partial \mathbf{f}_{K-1}[\mathbf{f}_{K-2}, \boldsymbol{\phi}_{K-1}]}{\partial \mathbf{f}_{K-2}} \cdots \frac{\partial \mathbf{f}_2[\mathbf{f}_1, \boldsymbol{\phi}_2]}{\partial \mathbf{f}_1} \cdot \frac{\partial \mathbf{f}_1[\mathbf{z}, \boldsymbol{\phi}_1]}{\partial \mathbf{z}}$$

Similarly, the Jacobian of the inverse mapping is the reciprocal of this.

4. We train the normalizing flow by maximum likelihood on data examples x_i, mapping them to z_i in the base distribution and optimizing the log-likelihood log|∂f/∂z| - log $p(z_i)$.

$$\underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \log\left[\left|\frac{\partial \mathbf{f}[\mathbf{z}_i, \boldsymbol{\phi}]}{\partial \mathbf{z}_i}\right|\right] - \log\left[Pr(\mathbf{z}_i)\right]\right]$$

So in summary, we compose invertible neural network layers to transform a simple base distribution to a complex one, with the Jacobian accounting for the distortions.

The Jacobian Determinant

Understanding the Jacobian Determinant

- It measures how a function distorts the space around a point
- Plays a crucial role in transforming probability distributions

The Jacobian Matrix

- Consider a function that maps a point from one space to another
- The Jacobian matrix describes how the function distorts the space around that point
- It contains information about stretching, squishing, or rotating the space in different directions

The Jacobian Determinant

The Jacobian determinant is a single number that summarizes the overall effect of the distortion

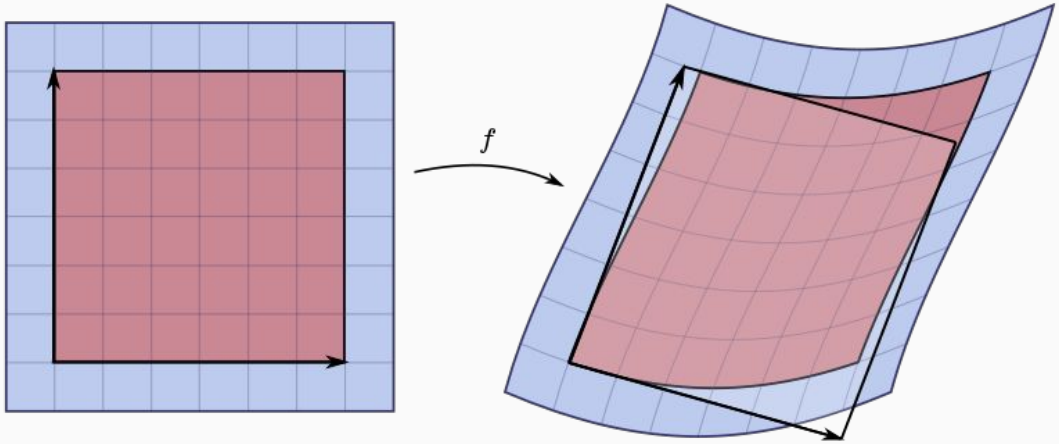
It tells you how much the function expands or shrinks the volume (or area in 2D) around the point

- Greater than 1: expanding the space, making the volume larger
- Between 0 and 1: contracting the space, making the volume smaller
- Exactly 1: preserving the volume, keeping it the same
- 0: collapsing the space, mapping it to a lower-dimensional space
- Negative: flipping the orientation of the space, turning it inside out

The Jacobian Determinant in Probability Distributions

- When transforming a random variable using a function, the Jacobian determinant helps adjust the probability density
- It accounts for the change in volume caused by the transformation
- This adjustment ensures that the total probability remains equal to 1
- Essential for maintaining a valid probability distribution

Example



A nonlinear map $f: R2 \rightarrow R2$

- Sends a small square (left, in red) to a distorted parallelogram (right, in red).
- The Jacobian at a point gives the best linear approximation of the distorted parallelogram near that point (right, in translucent white),
- and the Jacobian determinant gives the ratio of the area of the approximating parallelogram to that of the original square

https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant

The Jacobian Determinant in Normalizing Flows

- Normalizing flows models learn complex probability distributions
- The Jacobian determinant plays a key role in enabling normalizing flows to work effectively
- By designing transformations with easily computable Jacobian determinants, normalizing flows can:
    - Efficiently calculate the likelihood of the data
    - Generate new samples from the learned distribution

Normalizing flow model - decoding function

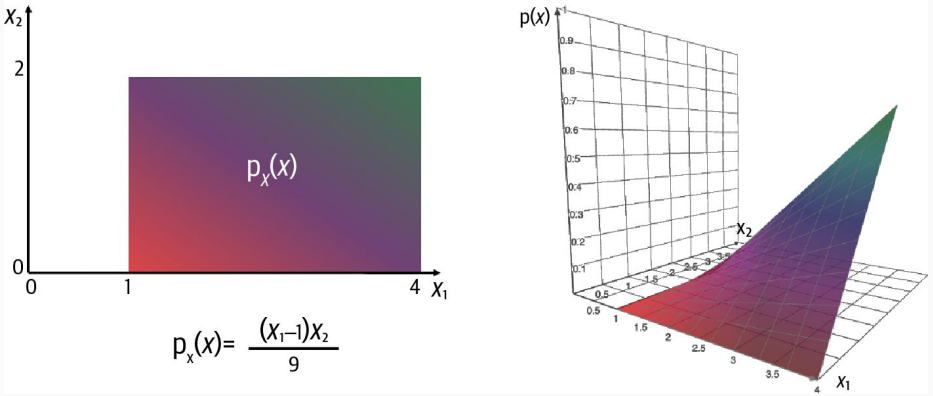
- In a normalizing flow model, the decoding function is designed to be the exact inverse of the encoding function and quick to calculate, giving normalizing flows the property of tractability.
- However, neural networks are not by default invertible functions.
- This raises the question of how we can create an invertible process that converts between a complex distribution and a much simpler distribution (such as a bell-shaped Gaussian distribution)
- while still making use of the flexibility and power of deep learning.

Change of variables

Probability distribution

- A probability distribution $p_X(x)$defined over two dimensions $x = (x_1, x_2)$, shown in 2D (left) and 3D (right).
- This function integrates to 1 over the domain of the distribution (i.e., in the range [1, 4] and in the range [0, 2])
- It represents a well-defined probability distribution.
- We can write this as follows:

$$\int_0^2 \int_1^4 p_X(x) dx_1 dx_2 = 1$$

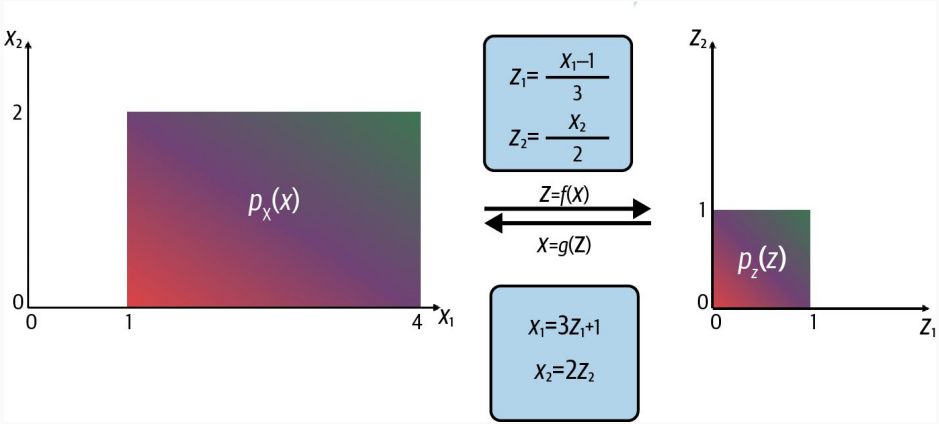


Shift and scale

- To shift and scale this distribution so that it is defined over a unit square *Z*.
- We define a new variable and a function that maps each point in to exactly one point in as follows:

$$
\begin{aligned}
z &= f(x) \\
z_1 &= \frac{x_1 - 1}{3} \\
z_2 &= \frac{x_2}{2}
\end{aligned}
$$

The shift and scale function



- This function is *invertible*.
- That is, there is a function *g* that maps every *z* back to its corresponding *x*.
- This is essential for a change of variables, as otherwise we cannot consistently map backward and forward between the two spaces.
- In conclusion, we can find *g* by rearranging the equations that define *f*.

Change of variables from X to Z

- We now need to see how the change of variables from X to Z affects the probability distribution $p_X(x)$.
- We can do this by plugging the equations that define g into $p_X(x)$ to transform it into a function $p_Z(z)$ that is defined in terms of z:

$$p_Z(z) = \frac{((3z_1 + 1) - 1)(2z_2)}{9} = \frac{2z_1 z_2}{3}$$

- However, if we now integrate $p_Z(z)$over the unit square, we can see that we have a problem!

$$\int_0^1 \int_0^1 \frac{2z_1 z_2}{3} dz_1 dz_2 = \frac{1}{6}$$

- The transformed function $p_Z(z)$ is now no longer a valid probability distribution, because it only integrates to 1/6.
- If we want to transform our complex probability distribution over the data into a simpler distribution that we can sample from, we must ensure that it integrates to 1

Jacobian determinant of the transformation

- The missing factor of 6 is due to the fact that the domain of our transformed probability distribution is six times smaller than the original domain
- The original rectangle X had area 6, and this has been compressed into a unit square Z that only has area 1.
- Therefore, we need to multiply the new probability distribution by a normalization factor that is equal to the relative change in area (or volume in higher dimensions).
- Luckily, there is a way to calculate this volume change for a given transformation—it is the absolute value of the Jacobian determinant of the transformation.

Jacobian of a function

- The *Jacobian* of a function $z=f(x)$ is the matrix of its first-order partial derivatives, as shown here:

$$\mathrm{J} = \frac{\partial z}{\partial x} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_1}{\partial x_n} \\ \ddots & \vdots & \\ \frac{\partial z_m}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_n} \end{bmatrix}$$

- If we take the partial derivative of
  - z_1 with respect to x_1, we obtain 1/3.
  - z_1 with respect to x_2, we obtain 0.
  - z_2 with respect to x_1, we obtain 0.
  - z_2 with respect to x_2, we obtain 1/2.
- Therefore, the Jacobian matrix for our function $f(x)$ is as follows:

$$J = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

Scaling factor

- The formula for calculating the determinant of a matrix with two dimensions is:

$$\det\begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

- For our example, the determinant of the Jacobian is

$$\frac{1}{3} \times \frac{1}{2} - 0 \times 0 = \frac{1}{6}$$

- This is the scaling factor of 1/6 that we need to ensure that the probability distribution after transformation still integrates to 1!

Change of Variables

1. We start with a probability distribution $p_X(x)$ defined over a space X (e.g. a rectangle).
2. We define a transform f that maps points x in X to points z in a new space Z (e.g. scaling to a unit square).
3. If we plug f into p_X, we get a function $p_Z(z)$ defined over Z.
4. However, $p_Z(z)$ is no longer a valid probability distribution - it won't integrate to 1 over Z.
5. This is because f has changed the volumes/areas that p_X was defined over.
6. To fix this, we need to multiply $p_Z(z)$ by a normalization factor equal to the relative volume change.
7. This normalization factor is given by the absolute Jacobian determinant $|J_f|$.
8. The Jacobian J_f is the matrix of all partial derivatives of f.
9. Its absolute determinant $|J_f|$ gives the local volume scaling factor for an infinitesimal region.
10. By multiplying by $|J_f|^{-1}$, we counteract the distortion f causes and p_Z integrates to 1.

So in summary, when transforming between spaces with probability distributions, we need to account for volume changes using the Jacobian determinant - this allows us to preserve normalization.

Change of variables equation

- The change of variables equation is: $p_X(x) = p_Z(z)\ |\det J_f|$

  Where:

  p_X is the complex data distribution

  p_Z is a simple distribution (e.g. Gaussian)

  f maps points z from the simple distribution to points x in the data distribution

  J_f is the Jacobian matrix (partial derivatives) of f

  $|\det J_f|$ is the absolute determinant of J_f
- To build a generative model:
  - We want to find an invertible mapping f that transforms points z from the simple p_Z into the data space p_X.
  - We then have an exact formula connecting p_Z and p_X using the Jacobian determinant.
- Key issues:
  - Computing determinants in high dimensions is very expensive $(O(n^3))$.
  - Need special architecture for f that ensures:
    - It is invertible
    - The determinant is tractable to compute

Properties of Neural Network layers used in Normalizing flows

1. Expressiveness - The layers collectively need to be able to transform the simple base distribution (typically a multivariate Gaussian) into a complex distribution that can model the actual data distribution.

2. Invertibility - Each layer must define a one-to-one mapping between inputs and outputs (a bijection) so that there is a unique inverse mapping.

3. Efficient Inverse - We need to be able to compute the inverse mapping of each layer efficiently, since we have to apply the inverse repeatedly during likelihood evaluation and training. There must be a closed-form solution or fast algorithm.

4. Efficient Jacobian Calculation - We need to be able to efficiently calculate the determinant of the Jacobian for the mapping defined by each layer. This is needed to account for the density distortions caused by the transformation.

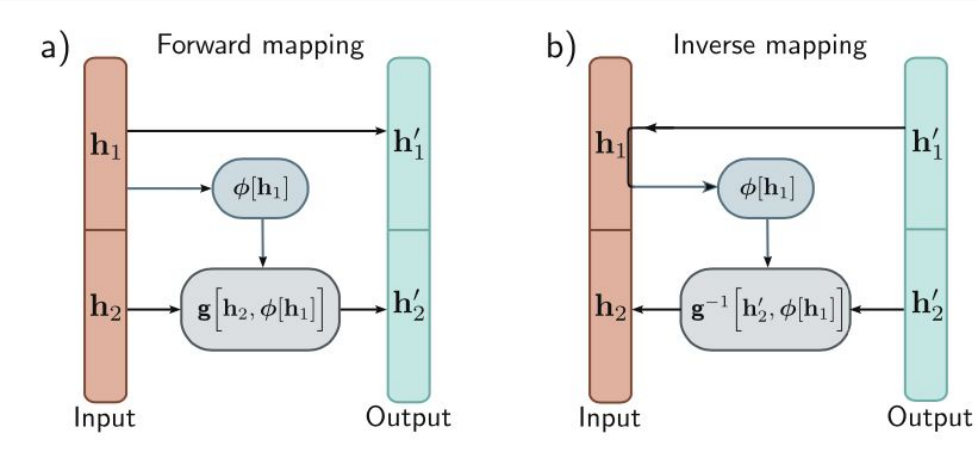
Satisfying all these desiderata allows normalizing flows to be used for efficient and expressive density estimation and generative modeling. The subsequent sections describe different types of invertible layers that can be used.

Coupling Flow

Coupling flows

- The input h is split into two parts h_1 and h_2.
- The first part of the output h'_1 is set equal to h_1 unchanged.
- The second part h'_2 is computed by applying an invertible transformation $g(h_2, \varphi)$ parameterized by φ.
- The parameters φ are output from a neural network that takes h_1 as input. So h'_2 depends on h_1 while h'_1 is unchanged.
- This is invertible - we can recover h_1 from h'_1 unchanged, then compute $\varphi(h_1)$ to invert g and recover h_2.
- The Jacobian is triangular so easy to compute.
- By splitting h differently and shuffling across layers, all variables are eventually transformed by the others.
- So in summary, coupling flows leave some input dimensions unchanged while transforming others based on the unchanged ones, in an invertible way. Iterating this allows flexible transformations.

Coupling flows



a) The input (orange vector) is divided into h_1 and h_2. The first part h'_1 of the output (cyan vector) is a copy of h_1. The output h'_2 is created by applying an invertible transformation $g[\bullet, \varphi]$ to h_2 , where the parameters φ are themselves a (not necessarily invertible) function of h_1.

b) In the inverse mapping, $h_1 = h'_1$. This allows us to calculate the parameters $\varphi[h_1]$ and then apply the inverse $g^{-1}[h'_2,\varphi]$ to retrieve h_2.

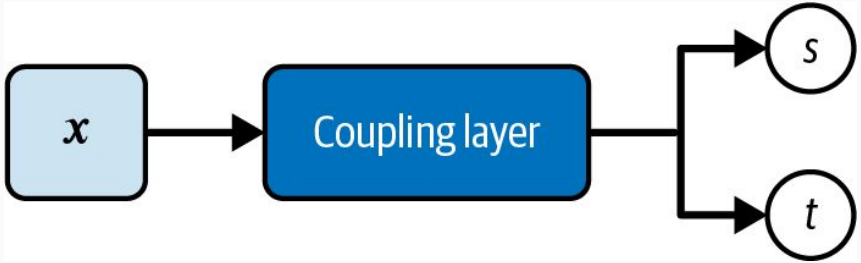
Real-valued non-volume preserving (real NVP) transformations

Coupling Layer - introduction

- Coupling Layer for Normalizing Flows:
  - Takes an input tensor 'x'
  - Applies a neural network consisting of multiple stacked Dense layers
- Generates two outputs:
  - Scaling Factor Tensor 's' of the same shape as 'x'
  - Translation Factor Tensor 't' of the same shape as 'x'
- The scaling and translation factors (s, t) are elementwise, allowing each element of 'x' to be transformed independently
- Adds flexibility and complexity to capture complex data distributions
- Stacking multiple Coupling layers with residual connections builds an expressive, invertible neural network model

Coupling Layers

A coupling layer outputs two tensors that are the same shape as the input: a scaling factor *(s)* and a translation factor *(t)*



TL; DR

- a Coupling layer uses stacked Dense layers to generate scaling and translation tensors of the same shape as the input tensor 'x', allowing each element to be transformed independently.
- This adds the modeling capacity to capture complex distributions when integrating multiple Coupling layers with residual connections.

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

Concept

Coupling Layers:

- Transform input data by splitting it into two parts.
- Each part is alternately processed and left unchanged using masks.

Purpose:

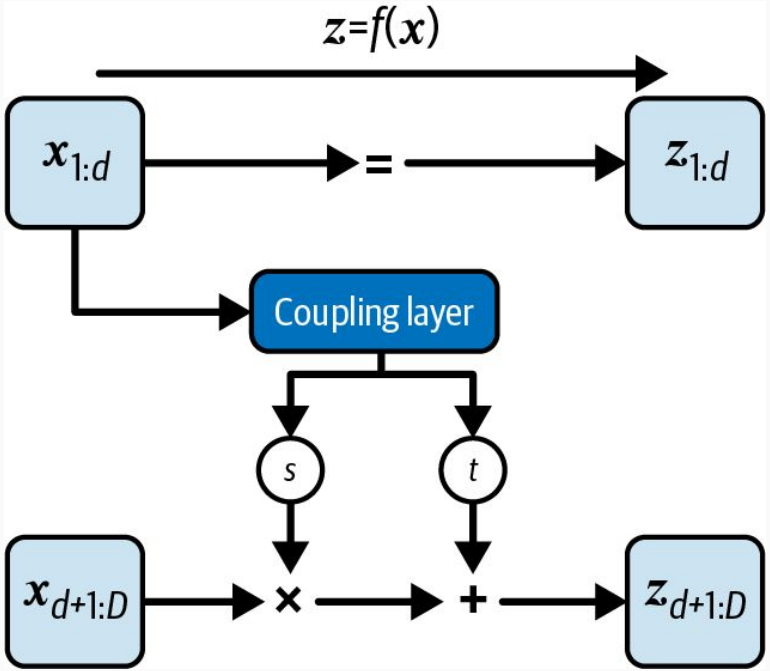
- Ensure efficient transformation of complex data.
- Maintain simplicity and invertibility in computations.

How Masking Works

Masking Process:

1. **Initial Mask:**
    - Splits input data into two parts: one is processed, the other remains unchanged.
    - Example: For input $x = [x_A, x_B]$:
        - Process x_B, keep x_A unchanged.
2. **Transformation:**
    - Compute scale (s) and translation (t) factors for the processed part.
    - Apply transformations: $z_B = x_B \cdot \exp(s(x_A)) + t(x_A)$.

The process of transforming the input *x* through a coupling layer



$$
\begin{aligned}
z_{1:d} &= x_{1:d} \\
z_{d+1:D} &= x_{d+1:D} \odot \exp\, s(x_{1:d})) + t\,(x_{1:d})
\end{aligned}
$$

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

Inverse Masking

- Purpose:
  - Ensure every element of input data is transformed over multiple layers.
- Mechanism:
  - Apply inverse mask to subsequent layers.
  - Process previously unchanged part, e.g., transform x_A in the next layer.

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

Jacobian matrix of the function

$$\frac{\partial z}{\partial x} = \begin{bmatrix} \mathbf{I} & 0 \\ \frac{\partial z_{d+1:D}}{\partial x_{1:d}} & \operatorname{diag}(\exp\left[s\left(x_{1:d}\right)\right]) \end{bmatrix}$$

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

Matrix Structure

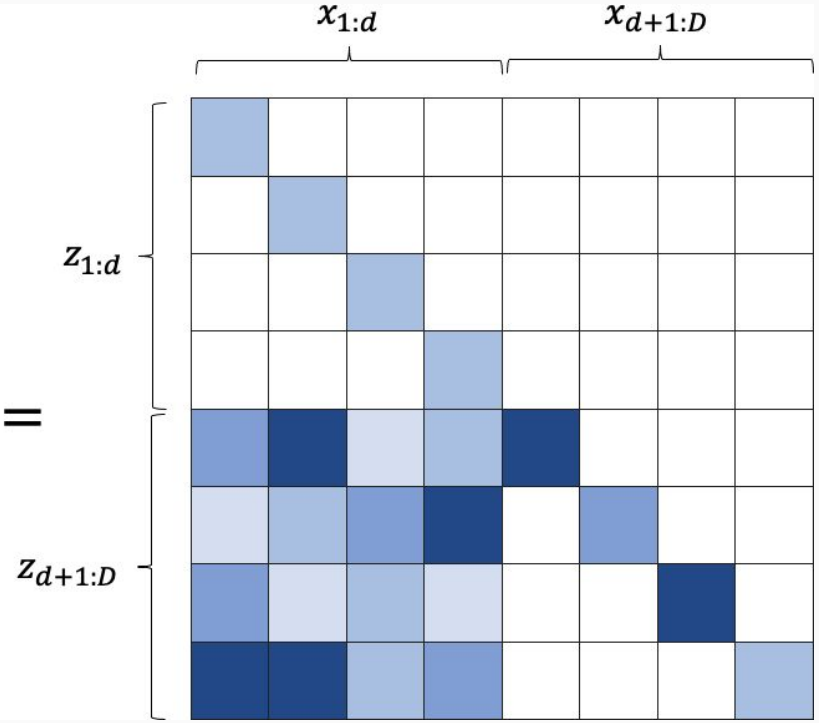
The top-left $d \times d$ submatrix is simply the identity matrix, because $z_{1:d} = x_{1:d}$. These elements are passed straight through without being updated. The top-right submatrix is therefore 0, because $z_{1:d}$ is not dependent on $x_{d+1:D}$.

The bottom-left submatrix is complex, and we do not seek to simplify this. The bottom-right submatrix is simply a diagonal matrix, filled with the elements of $\exp\left(s\left(x_{1:d}\right)\right)$, because $z_{d+1:D}$ is linearly dependent on $x_{d+1:D}$ and the gradient is dependent only on the scaling factor (not on the translation factor). Figure 6-7 shows a diagram of this matrix form, where only the nonzero elements are filled in with color.

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

The Jacobian matrix of the transformation—a lower triangular matrix

$$J = \frac{\partial z}{\partial x} =$$

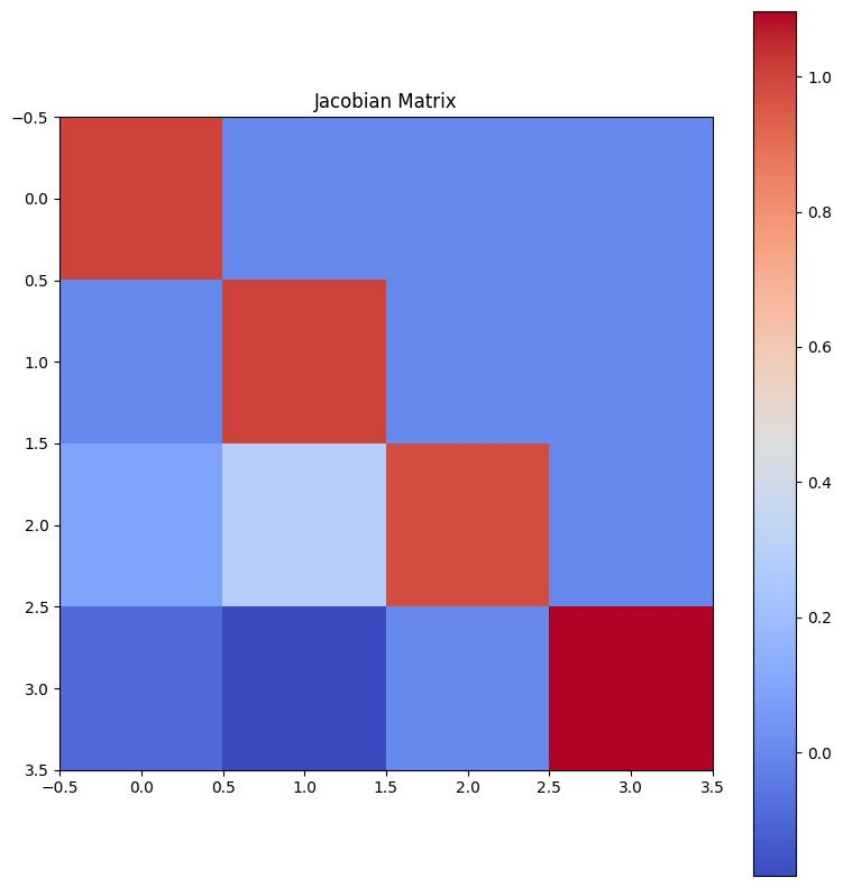


The Jacobian matrix of the transformation—a lower triangular matrix, with determinant equal to the product of the elements along the diagonal

- Lower triangular matrix.
- Top-left: Identity matrix (unchanged parts).
- Bottom-right: Diagonal matrix (dependent on scale factors).

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

The Jacobian matrix of the transformation—a lower triangular matrix



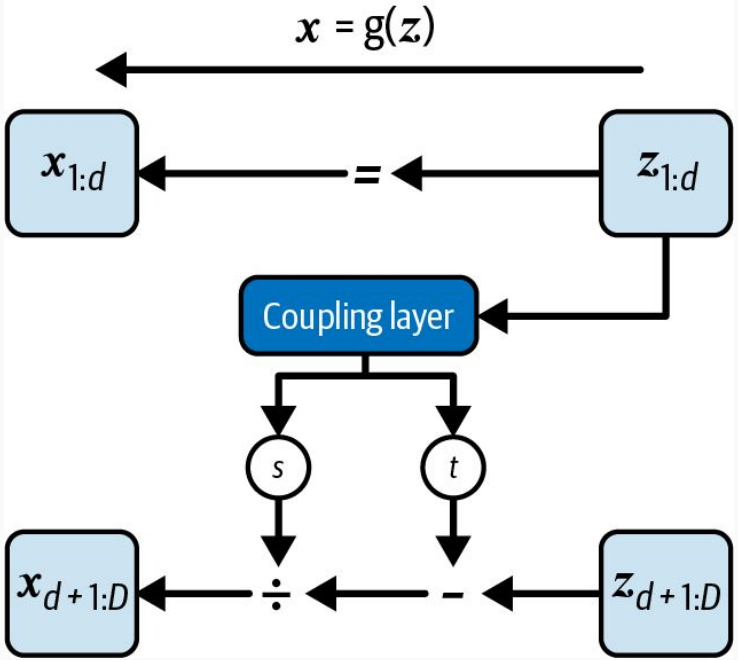
Why?

- The determinant of a lower-triangular matrix is just equal to the product of the diagonal elements.
- In other words, the determinant is not dependent on any of the complex derivatives in the bottom-left submatrix!
- This is easily computable, which was one of the two original goals of building a normalizing flow model.

$$\det\left(\mathrm{J}\right) = \exp\sum_{j} s(x_{1:d})_j \Bigg]$$

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

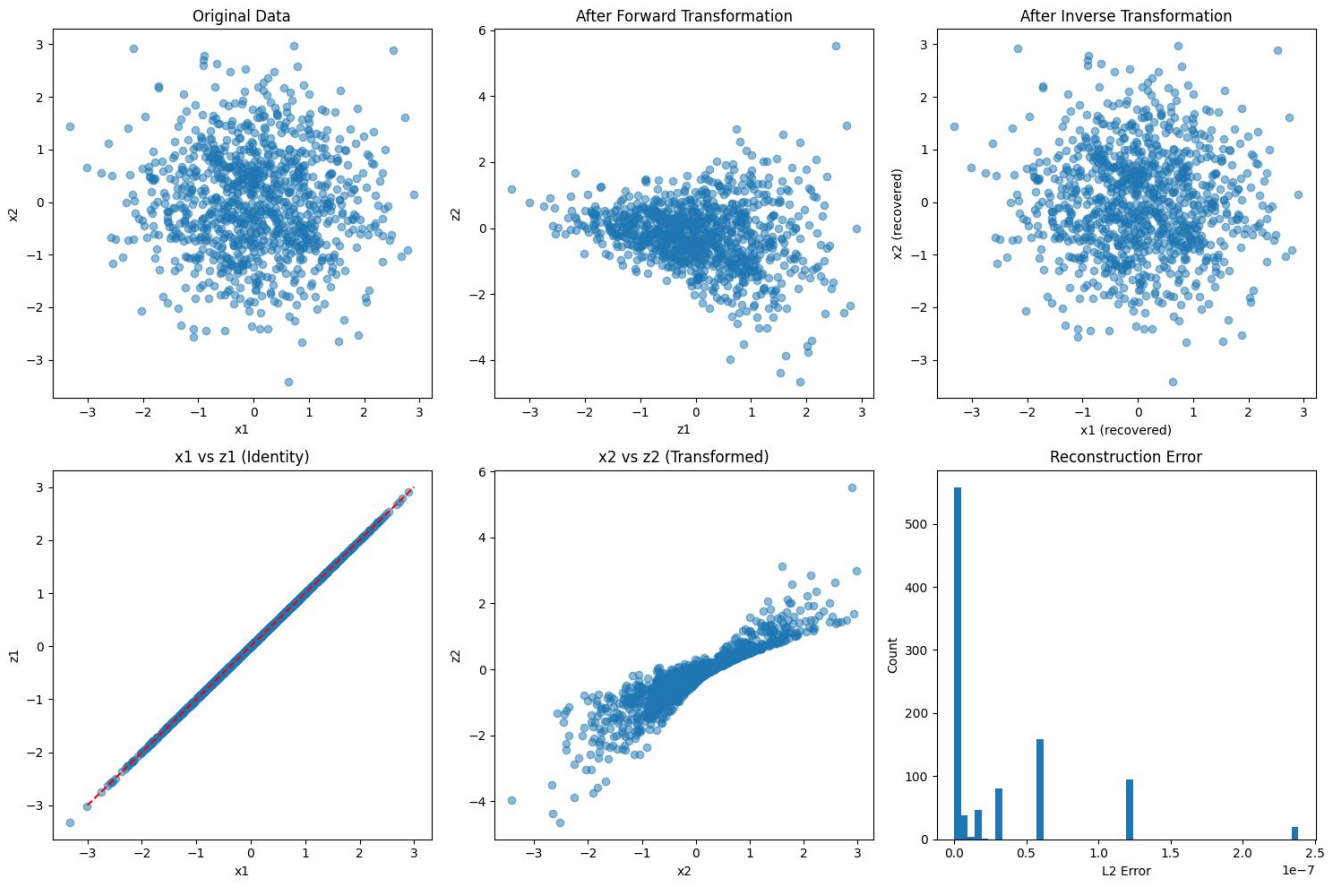
The inverse function $x = g(z)$



$$
\begin{aligned}
x_{1:d} &= z_{1:d} \\
x_{d+1:D} &= \left(z_{d+1:D} - t\left(x_{1:d}\right)\right) \odot \exp\left(-s\left(x_{1:d}\right)\right)
\end{aligned}
$$

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

How the coupling layer transforms the data



1. The first plot shows the original 2D data distribution.
2. The second plot shows the data after the forward transformation.
3. The third plot shows the data after applying the inverse transformation.
4. The fourth plot (x1 vs z1) shows that the first half of the input is unchanged.
5. The fifth plot (x2 vs z2) shows how the second half is transformed.
6. The last plot shows the reconstruction error, which should be very close to zero.

This demonstrates how the coupling layer transforms the data in a complex yet invertible way.

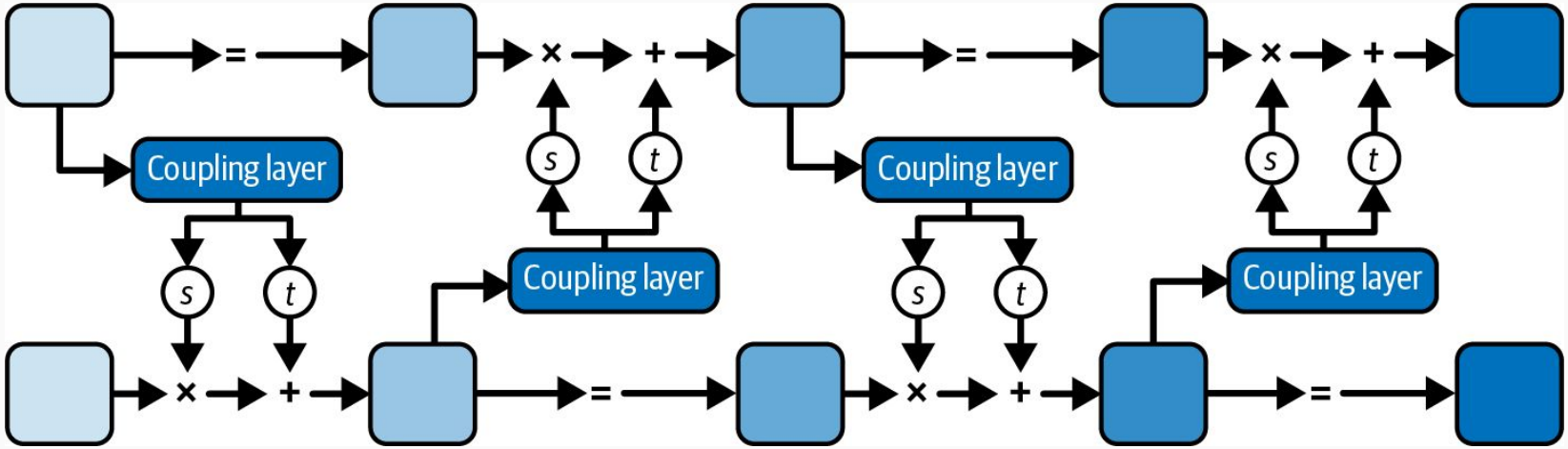
Problem: A single coupling layer leaves part of the data unchanged, limiting complexity capture.

Solution:

- Stack multiple coupling layers.
- Alternate the masking pattern for each layer.
- Each subsequent layer updates the previously unchanged part.

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

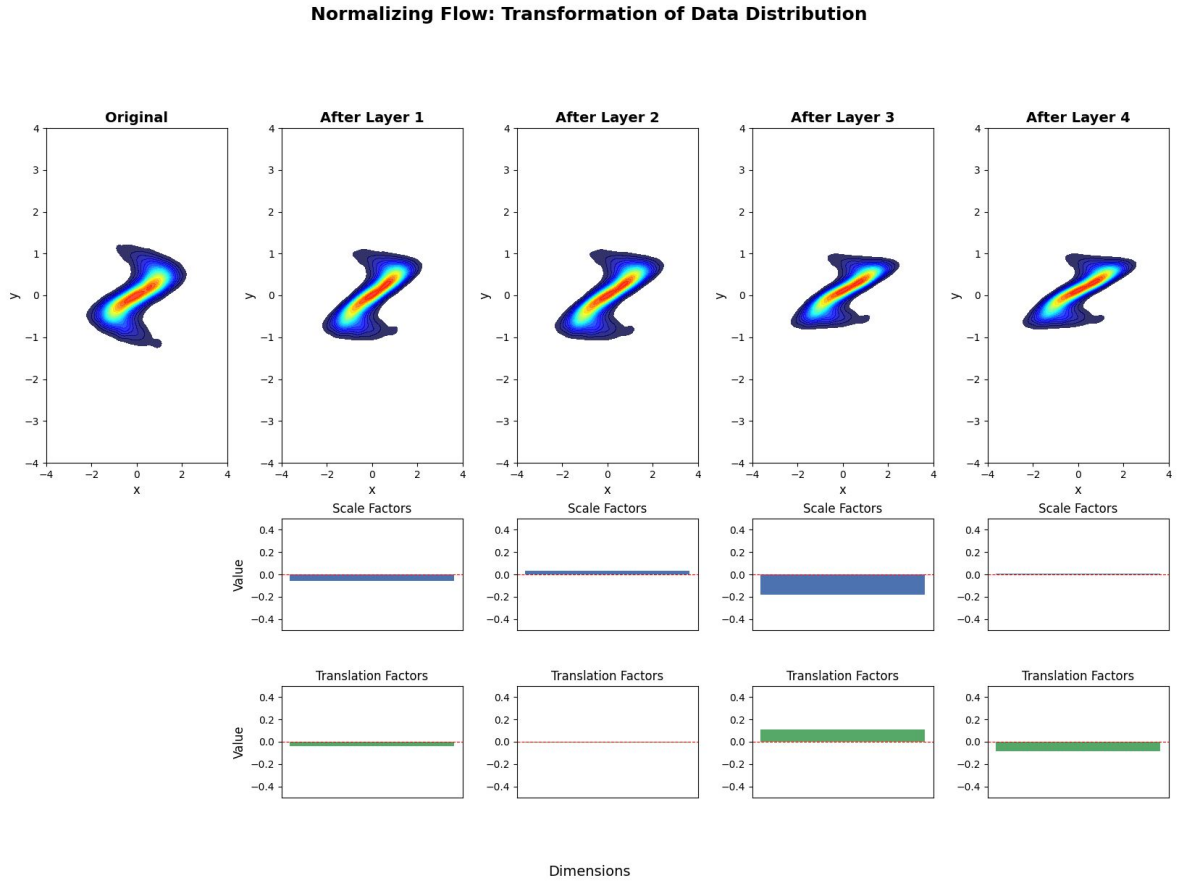
Stacking coupling layers



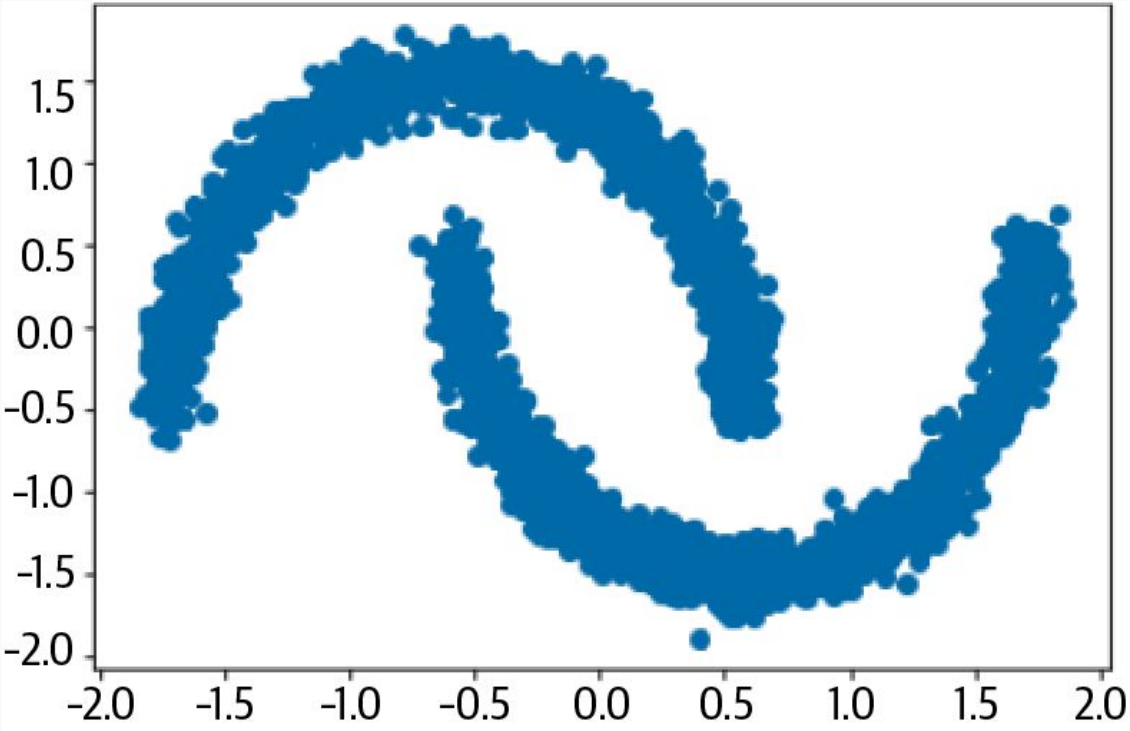
Stacking coupling layers, alternating the masking with each layer

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

Normalizing flow transforming a 2D data distribution

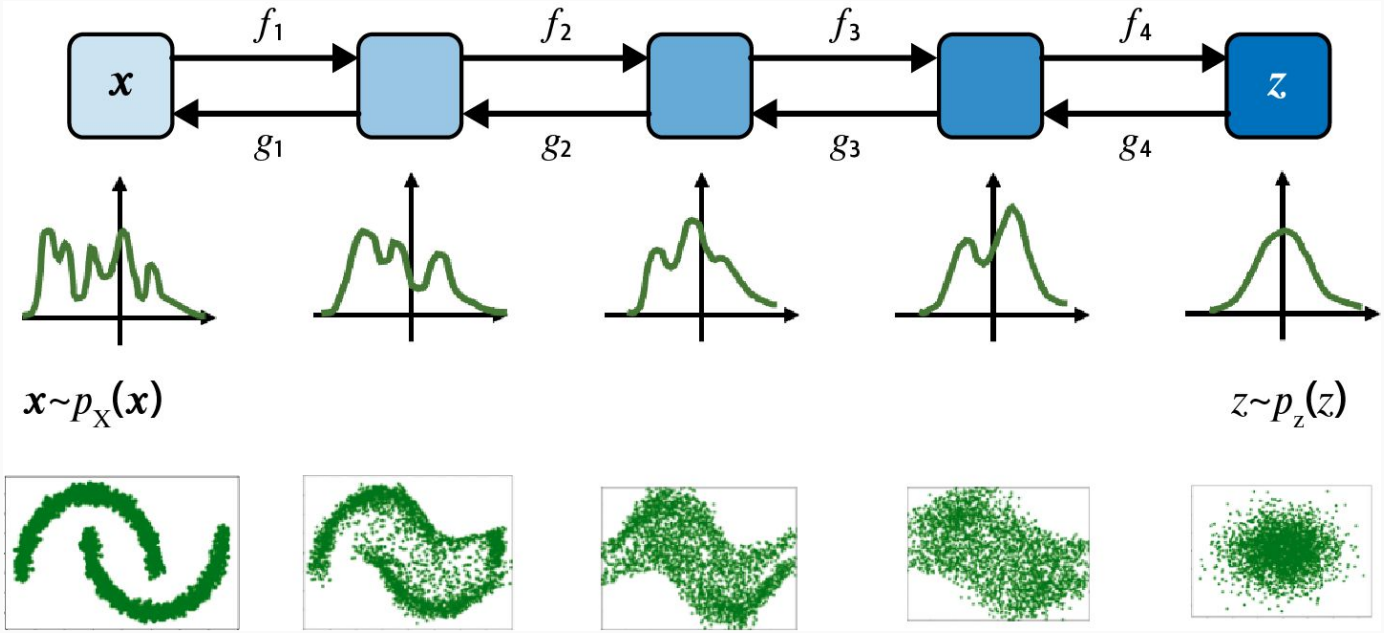


The Two Moons Dataset in two dimensions



Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

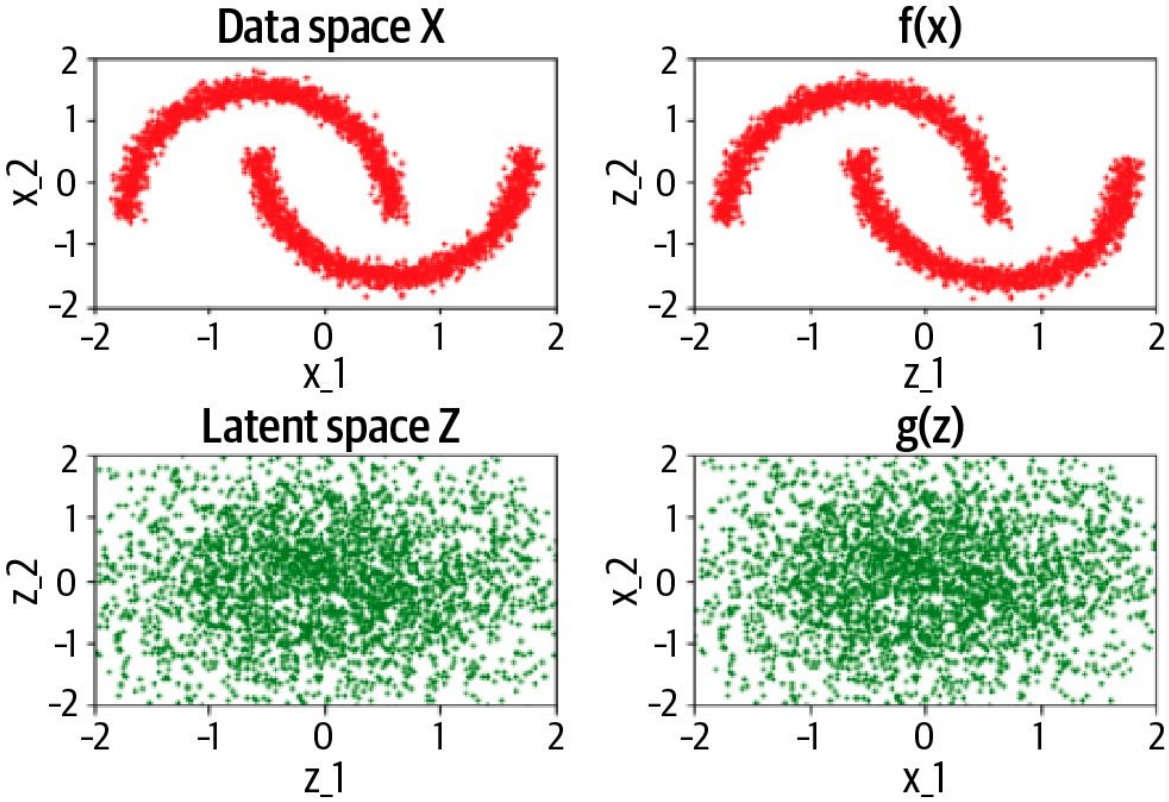
Training the RealNVP Model



Transforming between the complex distribution pX(x) and a simple Gaussian pZ(z) in 1D (middle row) and 2D (bottom row)

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

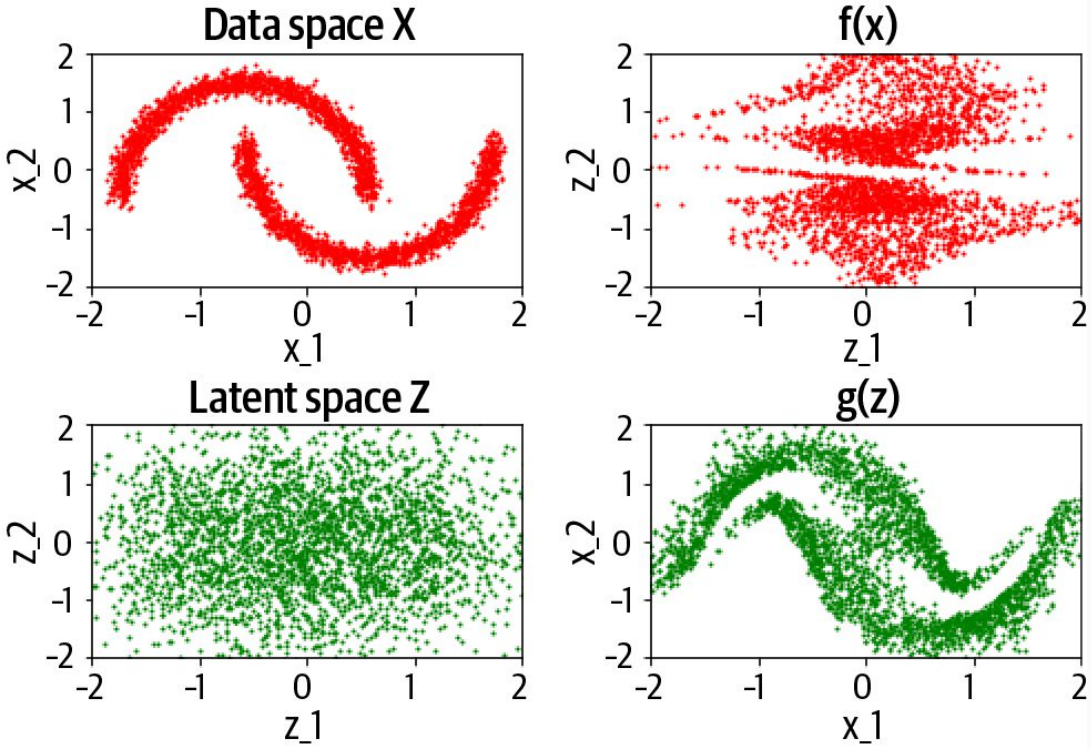
Analysis of the RealNVP Model



The RealNVP model inputs (left) and outputs (right) before training, for the forward process (top) and the reverse process (bottom)

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

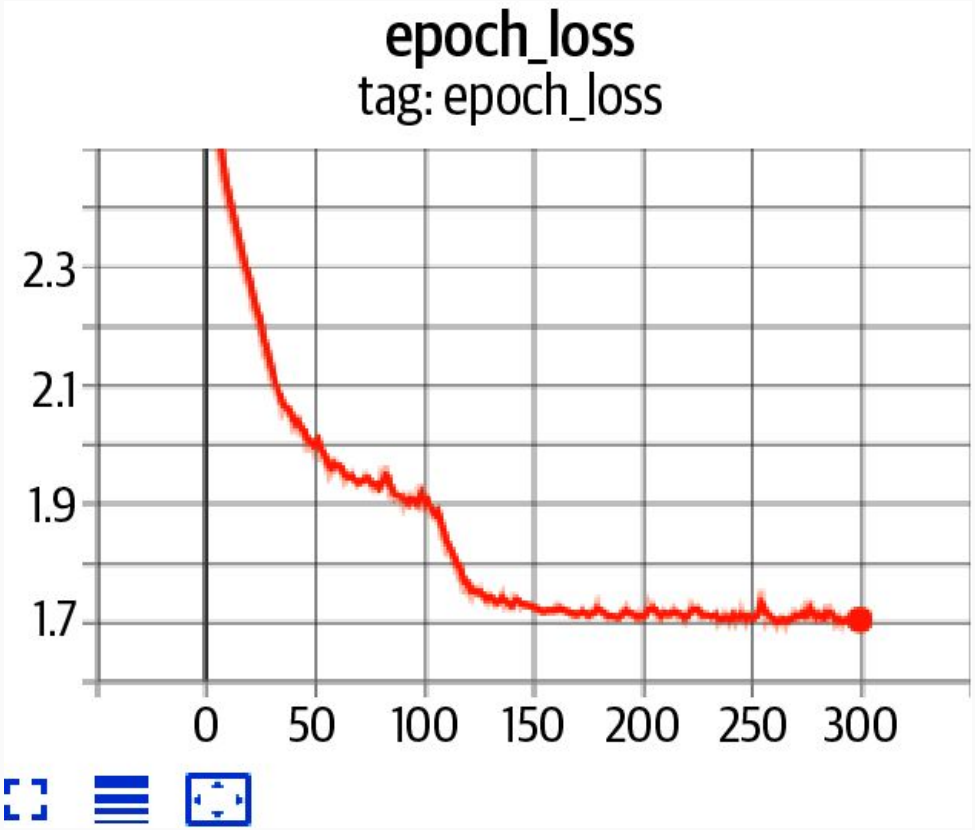
After Training



The RealNVP model inputs (left) and outputs (right) after training, for the forward process (top) and the reverse process (bottom)

Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

The loss curve for the RealNVP training process



Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc.

Connecting the dots - Constructing a Normalizing Flow Model

1. Choose base distribution (e.g. Gaussian) ➡ Simple, easy-to-sample distribution p(z)

2. Design invertible transformation layers (e.g. Coupling layers) ➡ Building blocks for complex, invertible transformations

3. Stack multiple layers, alternating patterns ➡ Expressive model capable of learning complex distributions

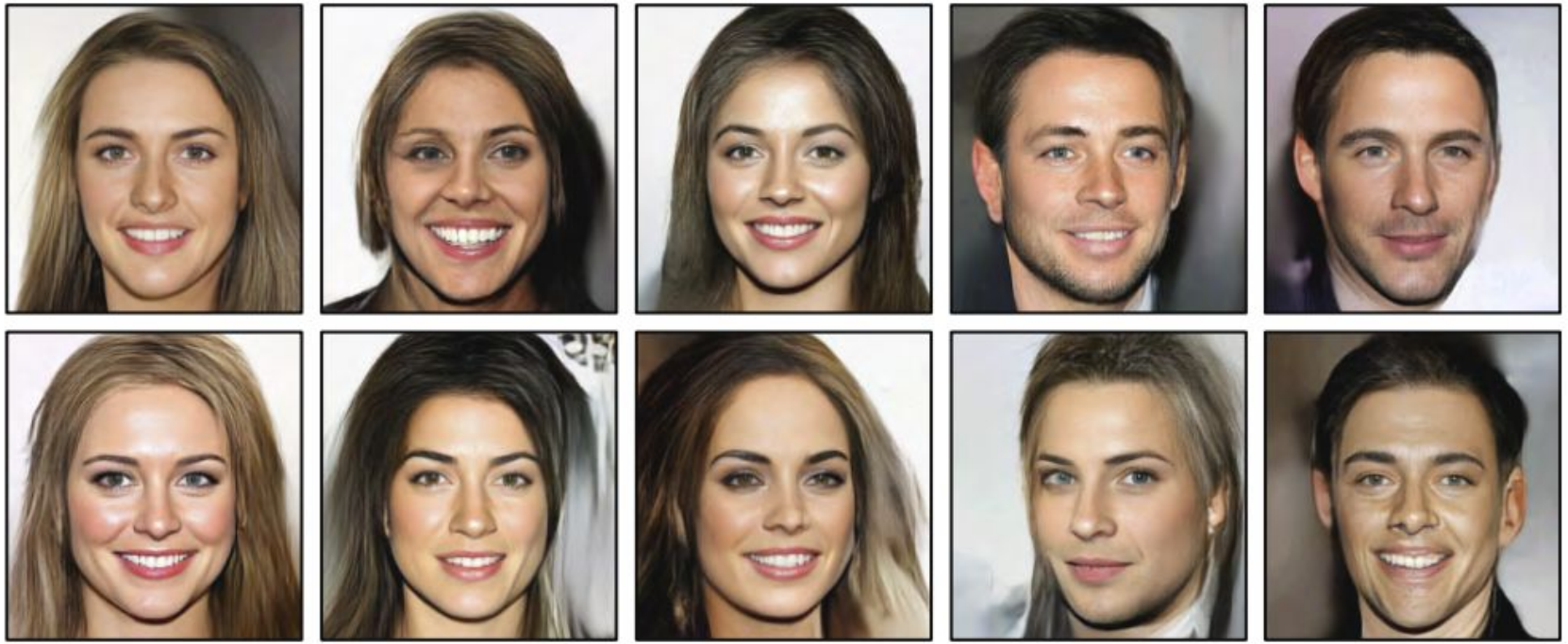
4. Ensure efficient Jacobian determinant computation ➡ Tractable likelihood calculation

5. Implement forward ($x = f(z)$) and inverse ($z = f^{-1}(x)$) mappings ➡ Bidirectional transformation between data and latent space

6. Define loss function: $\log p(x) = \log p(z) + \log |\det(\partial f/\partial z)|$ ➡ Objective for training the model

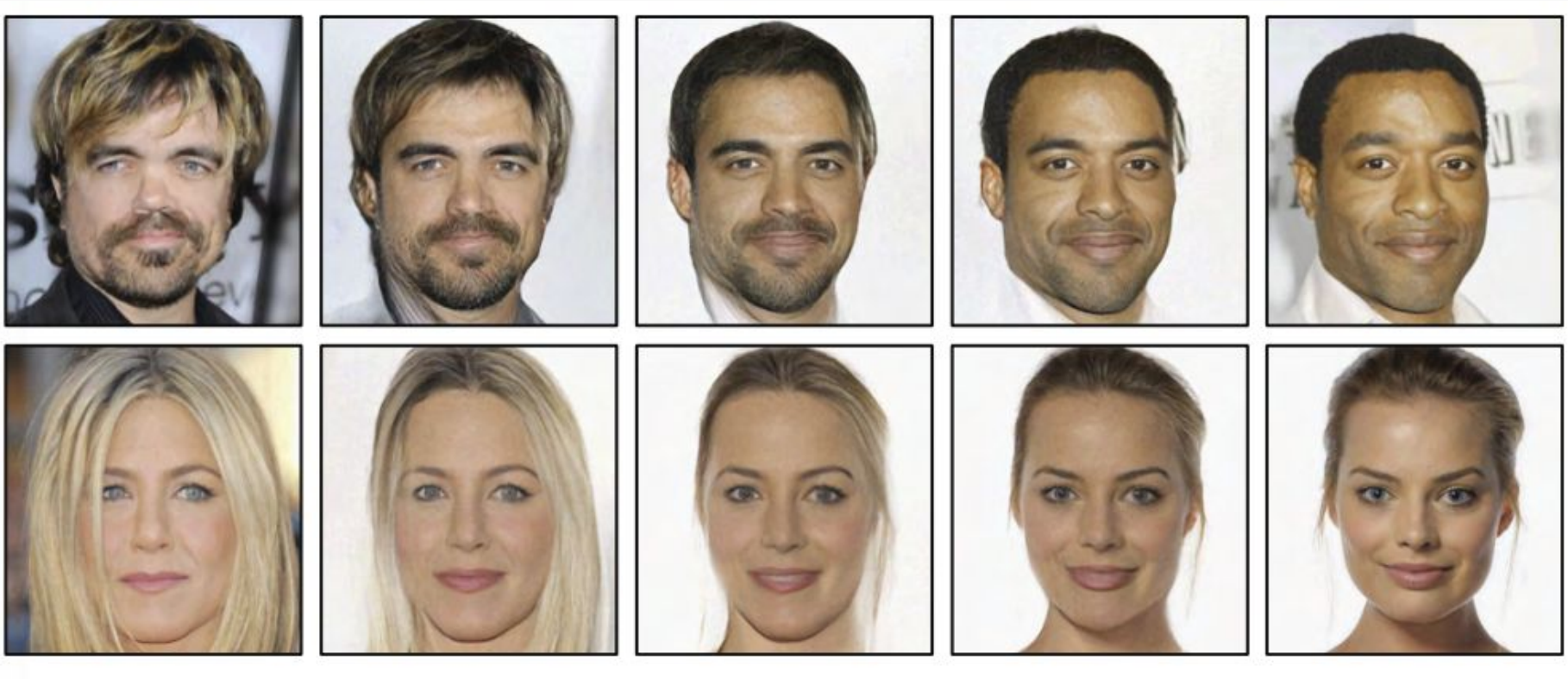
7. Train model by maximizing log-likelihood of training data ➡ Learned parameters that transform base distribution to target distribution

GLOW Model



Samples from GLOW trained on the CelebA HQ dataset (Karras et al., 2018). The samples are of reasonable quality, although GANs and diffusion models produce superior results. Adapted from Kingma & Dhariwal (2018).

Interpolation using GLOW model



The left and right images are real people. The intermediate images were computed by projecting the real images to the latent space, interpolating, and then projecting the interpolated points back to image space. Adapted from Kingma & Dhariwal (2018).

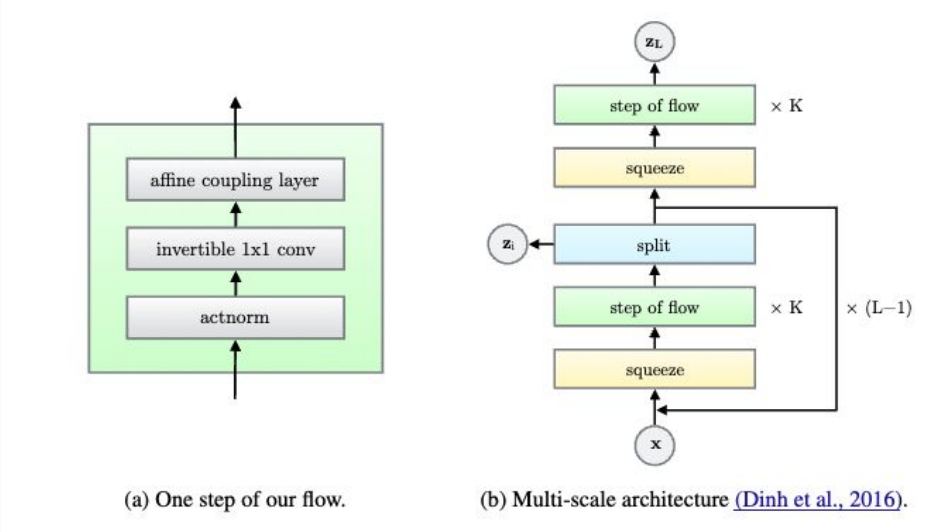
Key Components of Glow

- Actnorm Layer: Scale and bias with data-dependent initialization
- Invertible 1×1 Convolution: Permutation operation replaced by learnable convolution
- Affine Coupling Layer: Efficient reversible transformation for manipulating data
- Multi-Scale Architecture: Combines multiple flow steps for complex transformations

Kingma, Diederik P., and Prafulla Dhariwal. "Glow: generative flow with invertible 1× 1 convolutions." *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018.

Actnorm

- A generative flow where each step (left) consists of an actnorm step
- Followed by an invertible 1 × 1 convolution, followed by an affine transformation (Dinh et al., 2014).
- This flow is combined with a multi-scale architecture (right).



(a) One step of our flow.

(b) Multi-scale architecture (Dinh et al., 2016).

Kingma, Diederik P., and Prafulla Dhariwal. "Glow: generative flow with invertible 1× 1 convolutions." *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018.

ActNorm (Activation Normalization) in GLOW

Purpose: Normalize activations in normalizing flow models

Formula: $y = s * x + b$

- x: input, y: output
- s: learned scale, b: learned bias (per channel)

Features:

- Invertible: $x = (y - b) / s$
- Learnable parameters: s and b
- Data-dependent initialization
- Applied per-channel, constant across spatial dimensions

Advantages:

- Stabilizes training
- Maintains invertibility
- Suitable for variable batch sizes
- Efficient Jacobian computation

Quantitative Results

- Datasets Used: CIFAR-10, ImageNet, LSUN, CelebA HQ
- Performance Metrics: Bits per dimension (lower is better)
- Results: Significant improvement over RealNVP on standard benchmarks
- Efficiency: Faster convergence and efficient sampling

Kingma, Diederik P., and Prafulla Dhariwal. "Glow: generative flow with invertible 1× 1 convolutions." *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018.

Flows

Linear Flows

- A linear flow transforms the input h as $f(h) = \beta + \Omega h$, where Ω is a matrix.
- A linear flow is invertible if Ω is invertible. However, inverting and computing determinants for a general Ω scales poorly as $O(D^3)$ with dimensionality D.
- Special structures for Ω (diagonal, orthogonal, triangular) can reduce this cost, but reduce expressiveness.
- An efficient and still reasonably expressive parameterization is to construct Ω from the LU decomposition, with complexity $O(D^2)$.
- However, linear flows alone are not sufficiently expressive to transform a simple base distribution (e.g. Gaussian) to match an arbitrary complex distribution, since they preserve Gaussianity.
- In summary, linear flows provide a simple starting point for building normalizing flows, but more complex flows are needed to model real-world data distributions. The LU decomposition helps mitigate the computational issues.

Elementwise flows

- Elementwise flows apply the same scalar nonlinear function $f(h,\varphi)$ independently to each element h_d of the input vector h:

$$\mathbf{f}[\mathbf{h}] = \left[\mathrm{f}[h_1, \boldsymbol{\phi}], \mathrm{f}[h_2, \boldsymbol{\phi}], \ldots \mathrm{f}[h_D, \boldsymbol{\phi}]\right]^T$$

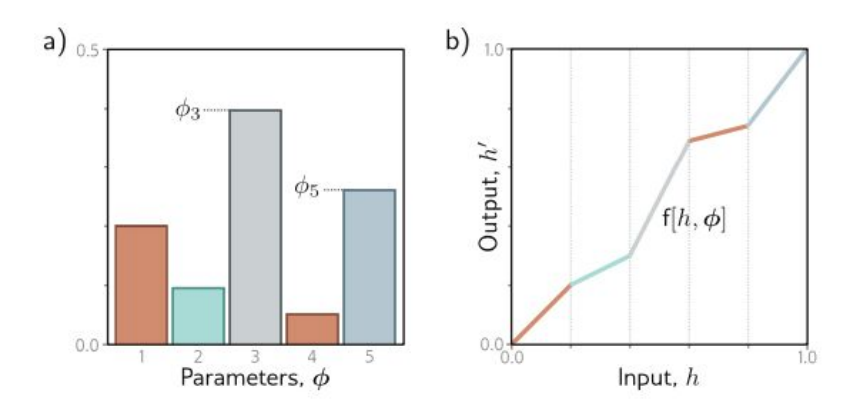
- This makes the Jacobian diagonal, so its determinant is just the product of the diagonal entries $\partial f/\partial h_d$. This is easy to compute.
- $f(h_d,\varphi)$ could be a fixed invertible nonlinearity like leaky ReLU, or a parameterized invertible function like a monotonic spline.
- Elementwise flows are nonlinear, but don't mix or create correlations between input dimensions.
- They are usually combined with linear flows that do mix dimensions. Elementwise flows also appear inside more complex flows.
- So in summary, elementwise flows provide an easy way to introduce invertible nonlinearities into normalizing flows, but other flows are needed to model correlations between variables. Their simple Jacobian helps tractability.

Example - Piecewise linear mapping

- An invertible piecewise linear mapping $h' = f[h, \phi]$ can be created by dividing the input domain $h \in [0, 1]$ into K equally sized regions *(here $K = 5$)*.
- Each region has a slope with parameter, ϕ_k.

  a) If these parameters are positive and sum to one, then

  b) the function will be invertible and map to the output domain $h' \in [0, 1]$



Autoregressive flows

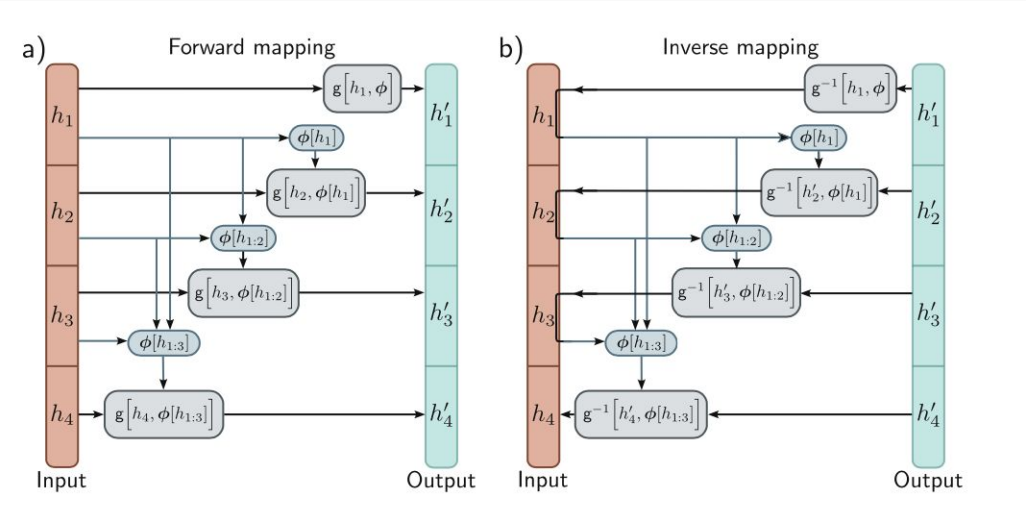
1. They transform each input dimension hd based on previous inputs $h_{1:d-1}$:

   $h'_d = g(h_d, \varphi[h_{1:d-1}])$

2. The transformer g( ) must be invertible, but the conditioner φ( ) can be any function, usually a neural network.
3. With sufficient flexibility, they can represent any distribution.
4. They can be computed in parallel using masking, called masked autoregressive flows.
5. However, inversion requires going sequentially from h_1 to h_D. So sampling is slow, though density estimation is fast.
6. An inverse autoregressive flow flips this - fast sampling but slow density estimation.
7. A trick is to train a fast inverse flow to mimic a masked flow which learns the density. The masked flow acts as the teacher.

In summary, autoregressive flows are very flexible but inversion/sampling is slow. Masking and using the flows in complementary directions alleviates the issues.

Example: Autoregressive flows



The input h (orange column) and output h' (cyan column) are split into their constituent dimensions (here four dimensions).

a) Output h'_1 is an invertible transformation of input h_1. Output h'_2 is an invertible function of input h_2 where the parameters depend on h_1. Output h'_3 is an invertible function of input h_3 where the parameters depend on previous inputs h_1 and h_2, and so on. None of the outputs depend on one another, so they can be computed in parallel.

b) The inverse of the autoregressive flow is computed using a similar method as for coupling flows. However, notice that to compute h_2 we must already know h_1, to compute h_3, we must already know h_1 and h_2, and so on. Consequently, the inverse cannot be computed in parallel.

Residual flows

1. The input h is split into h_1 and h_2, similar to coupling flows.
2. The output h'_1 is computed as h_1 plus a residual f1(h2,φ1) that depends on h2.
3. The output h'2 is h2 plus a residual $f_2(h'_1,\varphi_2)$ depending on h'_1.
4. This is invertible by simply reversing the order and turning the additions into subtractions:

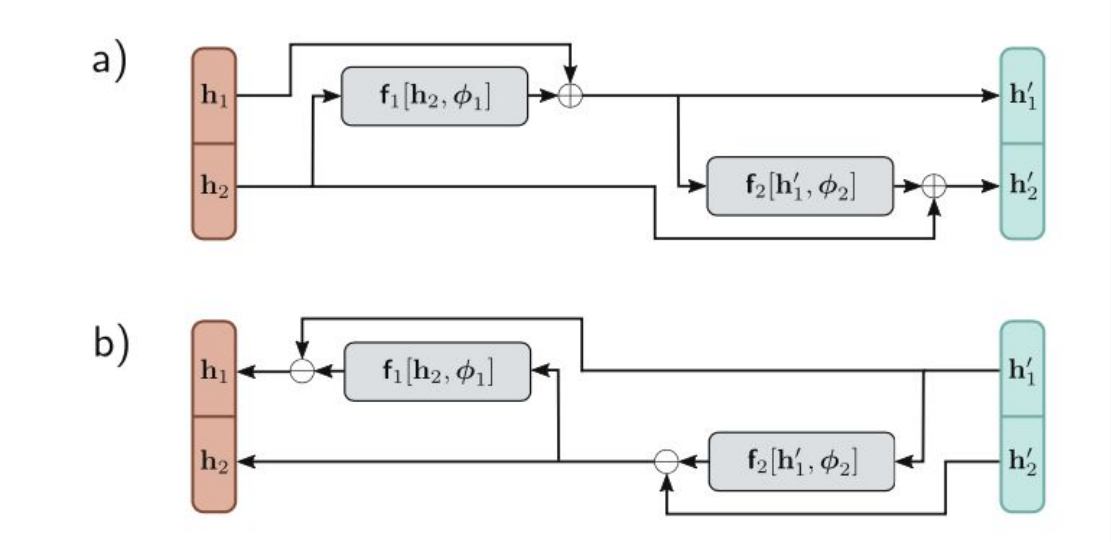
   $h_2 = h'_2 - f_2(h'_1,\varphi_2)$

   $h_1 = h'_1 - f_1(h2,\varphi1)$

5. There is no constraint for f_1 and f_2 to be invertible. However, the Jacobian determinant cannot be efficiently computed in the general case.
6. As with coupling flows, the inputs are permuted across layers so all variables interact.

So in summary, residual flows add invertibility while allowing unconstrained residual functions, but tractable likelihood calculation is lost in the process.

Example: Residual Flows



a) An invertible function is computed by splitting the input into h_1 and h_2 and creating two residual layers. In the first, h_2 is processed and h_1 is added. In the second, the result is processed, and h_2 is added.

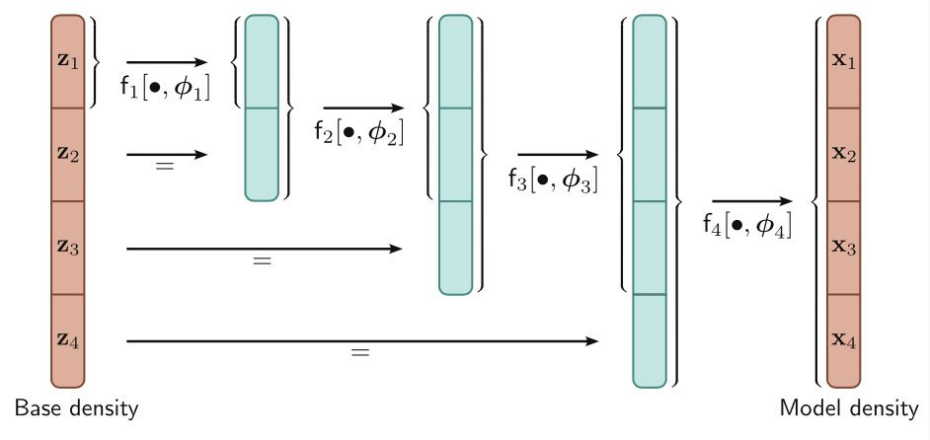
b) In the reverse mechanism the functions are computed in the opposite order, and the addition operation becomes subtraction

Contraction mappings in residual flows

1. The Banach fixed point theorem states that iteratively applying a contraction mapping f(z) (where distances shrink by a factor $\beta<1$) converges to a fixed point z = f(z).
2. This can be used to invert equations of the form y = z + f(z) by iterating z <- y - f(z) to find the z that maps to y.
3. Similarly, residual network layers h' = h + f(h) can be inverted if f(h) is a contraction mapping (Slope < 1 everywhere).
4. To ensure this, the weight matrices of f(h) can be clipped to small values.
5. The Jacobian log determinant can be approximated by viewing $I + \partial f/\partial h$ as a matrix exponential and using Hutchinson's trace estimator.

So in summary, the contractive nature of residual networks under certain conditions allows invertibility, which lets them be used as flows while still being unconstrained. But the Jacobian is more difficult to compute.

Multiscale flows



- The latent space z must be the same size as the model density in normalizing flows. However, it can be partitioned into several components, which can be gradually introduced at different layers.
- This makes both density estimation and sampling faster.
- For the inverse process, the black arrows are reversed, and the last part of each block skips the remaining processing.
- For example, $f^{-1}[\bullet,\varphi]$ only operates on the first three blocks, and the fourth block becomes z_4 and is assessed against the base density.

Modeling densities and image synthesis with normalizing flows

Density Modeling:

- Normalizing flows can compute exact log-likelihoods for samples, unlike GANs, VAEs, and diffusion models.
- This is useful for density estimation and anomaly detection - identifying low probability examples.

Image Synthesis (e.g. with GLOW):

- Uses a multi-scale architecture and coupling flows for invertibility.
- Periodically reduces resolution and removes channels into the latent code.
- Adds noise during training (dequantization) since images are discrete.
- Samples from the base density raised to a power for more realistic examples.
- Quality is not yet on par with GANs/diffusion models, perhaps due to architectural constraints.

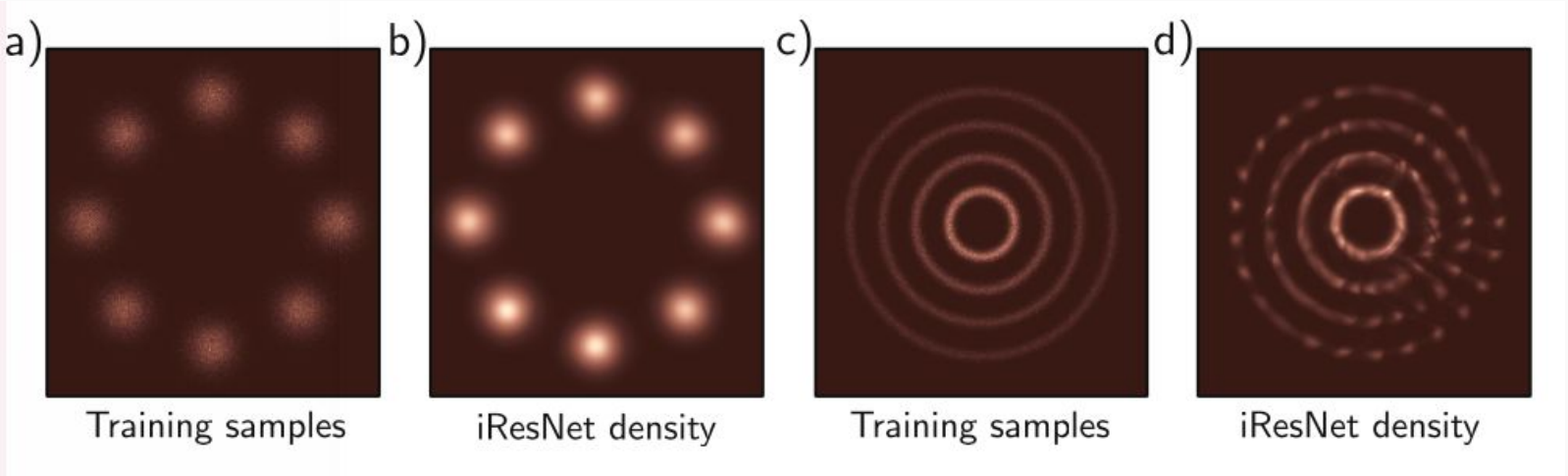
Properties:

- Can interpolate between real images by interpolating in the latent space and inverting.
- Provides control over high-level attributes via the latent code.

So in summary, normalizing flows allow explicit density modeling, with reasonable but not state-of-the-art sample quality currently.

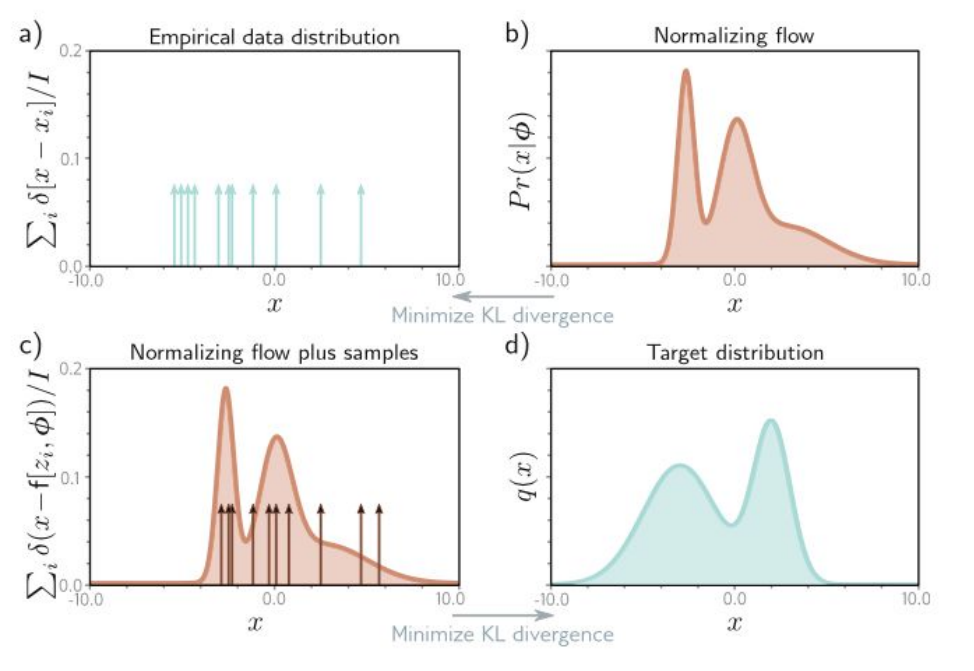
Their invertibility grants useful properties like interpolation.

Modeling densities



a) Toy 2D data samples. b) Modeled density using iResNet. c–d) Second example. Adapted from Behrmann et al. (2019)

Approximating density models



a) Training data.

b) Usually, we modify the flow model parameters to minimize the KL divergence from the training data to the flow model. This is equivalent to maximum likelihood fitting.

c) Alternatively, we can modify the flow parameters φ to minimize the KL divergence from the flow samples $x_i = f[z_i, \varphi]$ to a target density.

Speech Synthesis (Text-to-Speech)

- **Applications**: Flow-based models are used in **speech synthesis**, where the task is to generate human-like speech from text inputs. Models like **WaveGlow** use normalizing flows to generate high-quality, natural-sounding speech.
- **Everyday Use**: **Text-to-speech systems** (e.g., Google Assistant, Siri, Alexa) rely on deep generative models to convert text into lifelike speech that interacts with users in real-time

Text → Linguistic Features → Flow-based Model → Speech Output

Normalizing Flows in Speech Generation

- **Normalizing Flows** in models like WaveGlow start with **Gaussian noise**.
- Each step applies **invertible transformations** to convert noise into a speech waveform.
- The flow-based model gradually transforms noise into a **high-fidelity audio** output, ensuring **real-time** generation.

Key Steps:

1. Start with Gaussian noise.
2. Apply series of transformations (flows).
3. Output the speech waveform.

Benefits of Flow-Based Models for TTS

- **Real-Time Responses**: Flow-based models generate speech in parallel, reducing lag.
- **Natural Sounding Speech**: Models capture detailed speech patterns (intonation, rhythm) for lifelike voices.
- **Efficient on Devices**: Can run on smartphones and assistants without heavy computational demands.
- **Siri**, **Google Assistant**, and **voice-enabled systems** respond in natural, human-like speech.

References

- Foster, D. (2022). *Generative deep learning*. " O'Reilly Media, Inc. - Chapter 6
- *Prince, S. J. (2023). Understanding Deep Learning. MIT press.* - Chapter 16
- Kingma, Diederik P., and Prafulla Dhariwal. "Glow: generative flow with invertible 1× 1 convolutions." *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018.