

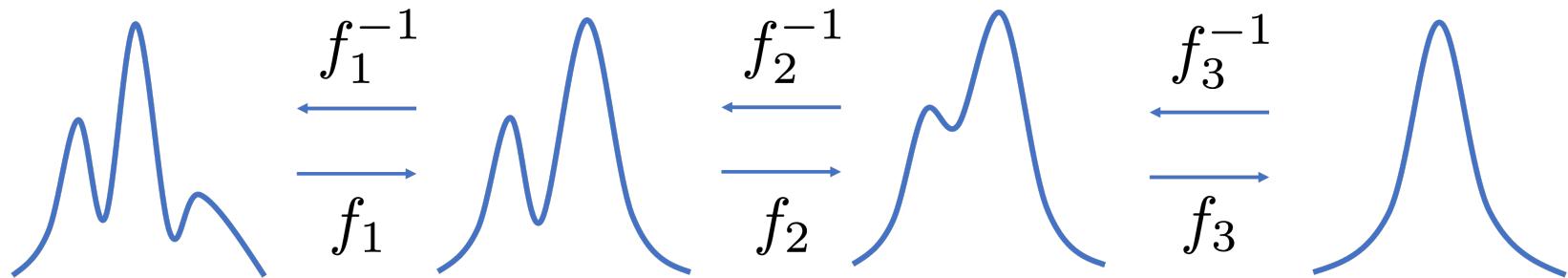
## 4. Normalizing Flows

M. Ravasi

AI Summer School @ KAUST

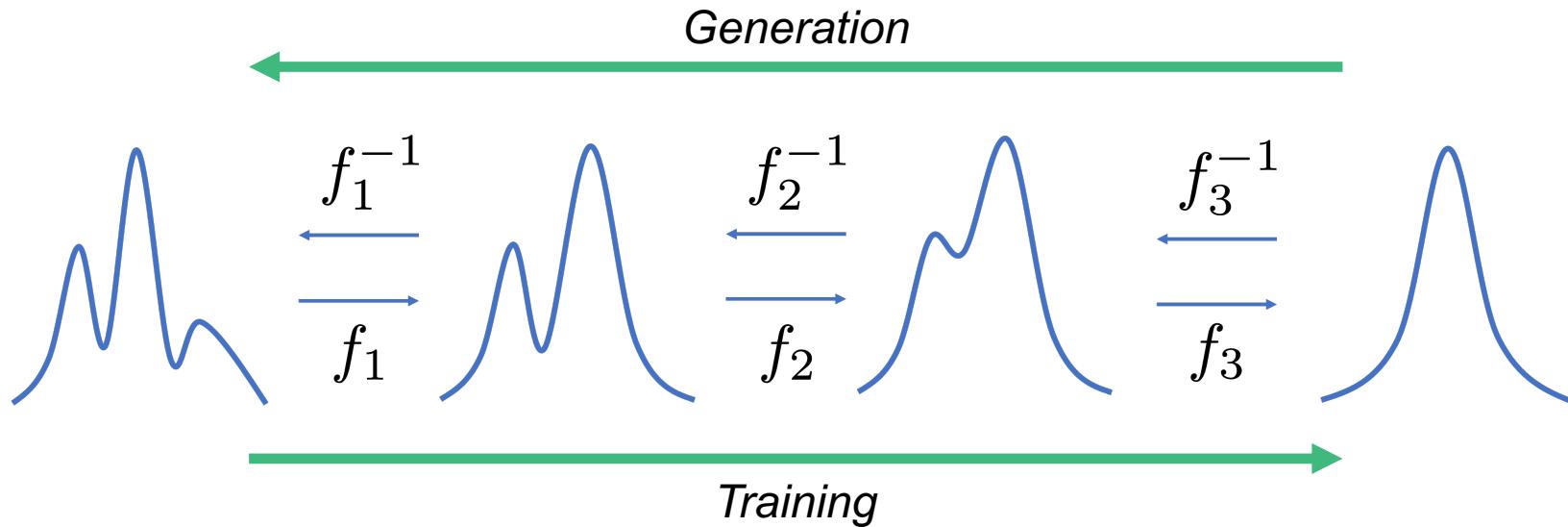
# Normalizing Flows: intuition

*Family of neural networks that can learn to **warp a simple density function** (e.g., gaussian) **into a complex one** (the one of interest). Provide tractable maximum likelihood estimation.*



# Normalizing Flows: intuition

*Family of neural networks that can learn to **warp a simple density function** (e.g., gaussian) **into a complex one** (the one of interest). Provide tractable maximum likelihood estimation.*



# Normalizing Flows: intuition

---

*Family of neural networks that can learn to warp a simple density function (e.g., gaussian) into a complex one (the one of interest). Provide **tractable maximum likelihood estimation.***

$\operatorname{argmax}_{\theta} p_{\theta}(\mathbf{x})$  is feasible without approximations

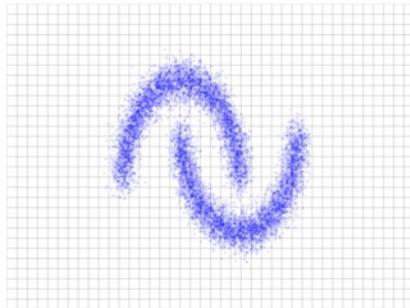
# Normalizing Flows: intuition

## Training

$$x \sim \hat{p}_X$$

$$z = f(x)$$

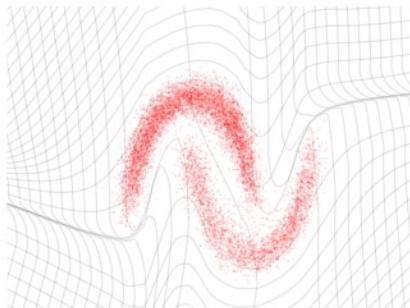
Data space  $\mathcal{X}$



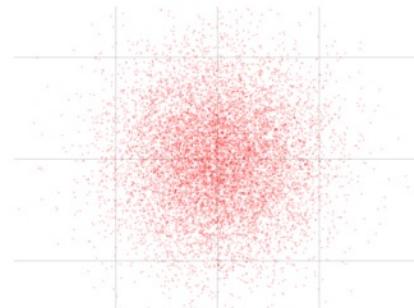
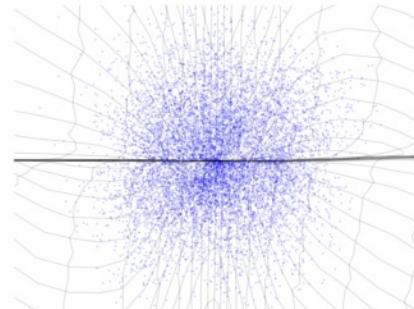
## Generation

$$z \sim p_Z$$

$$x = f^{-1}(z)$$

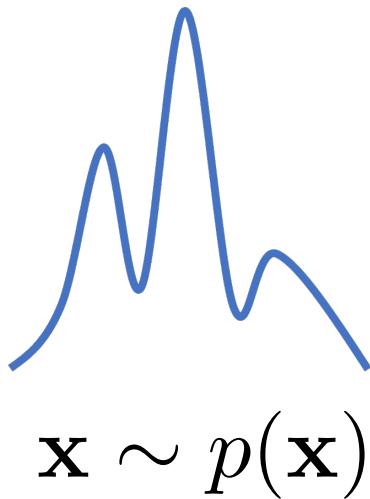


Latent space  $\mathcal{Z}$



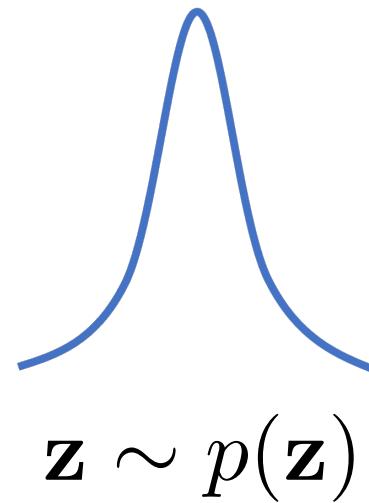
Dinh et al. (2014)

# Normalizing Flows: basic principle

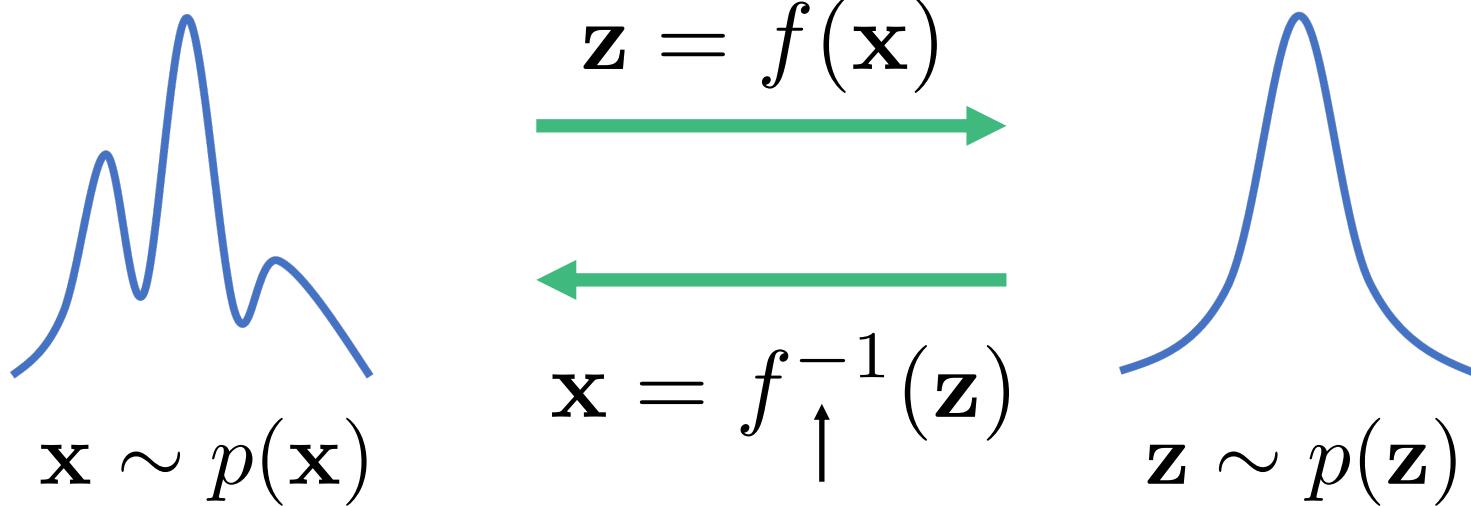


$$\mathbf{z} = f(\mathbf{x})$$
$$\mathbf{x} = f^{-1}(\mathbf{z})$$

The diagram illustrates the basic principle of a normalizing flow. It shows two probability density functions: a complex, multi-modal distribution  $\mathbf{x} \sim p(\mathbf{x})$  on the left, and a simple, unimodal distribution  $\mathbf{z} \sim p(\mathbf{z})$  on the right. Between them is a function  $f$ , represented by a blue curve that maps the complex distribution to the simple one. Two green arrows indicate the flow of data: a top arrow pointing right labeled  $\mathbf{z} = f(\mathbf{x})$  and a bottom arrow pointing left labeled  $\mathbf{x} = f^{-1}(\mathbf{z})$ .



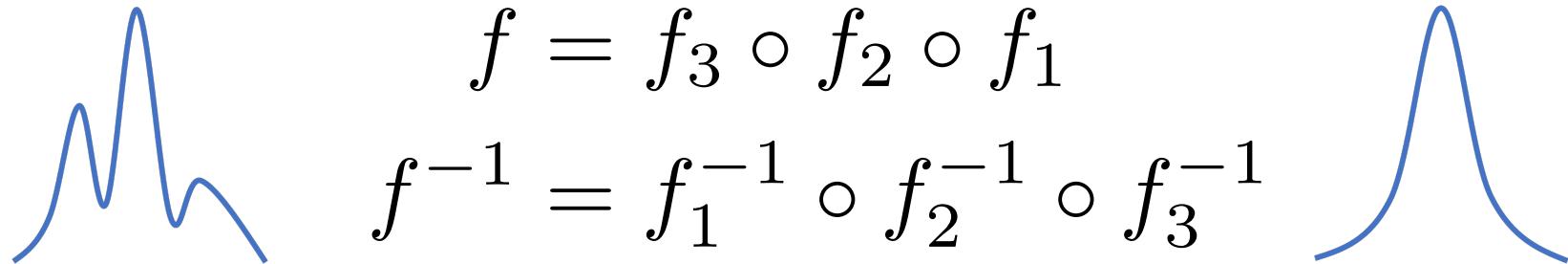
# Normalizing Flows: basic principle



Design an invertible function: we will  
need to introduce **Invertible NNs**

# Normalizing Flows: basic principle

In practice multiple transformations are required:

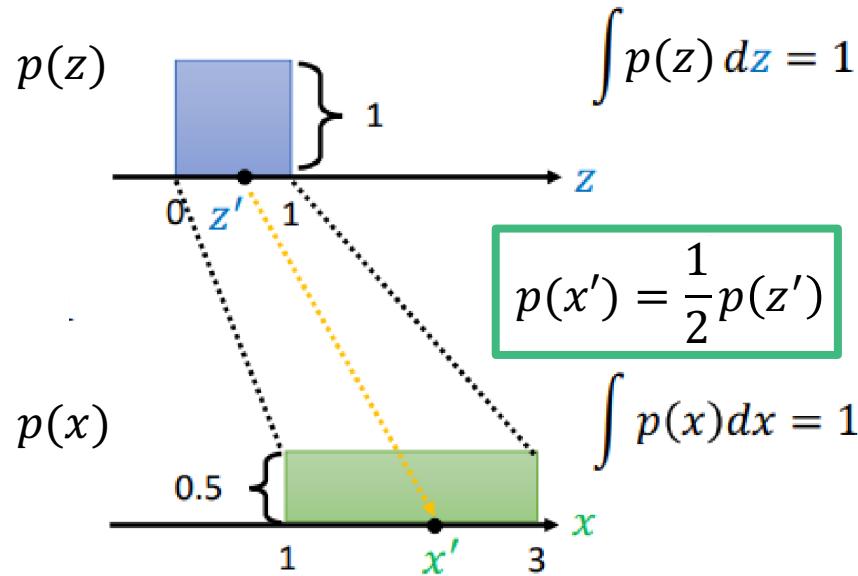


# Change of variable theorem

Define density of a random variable that is a deterministic transformation of another variable → **ensures still sums to 1!**

$$z = f(x) = \frac{x - 1}{2}$$

$$x = f^{-1}(z) = 2z + 1$$

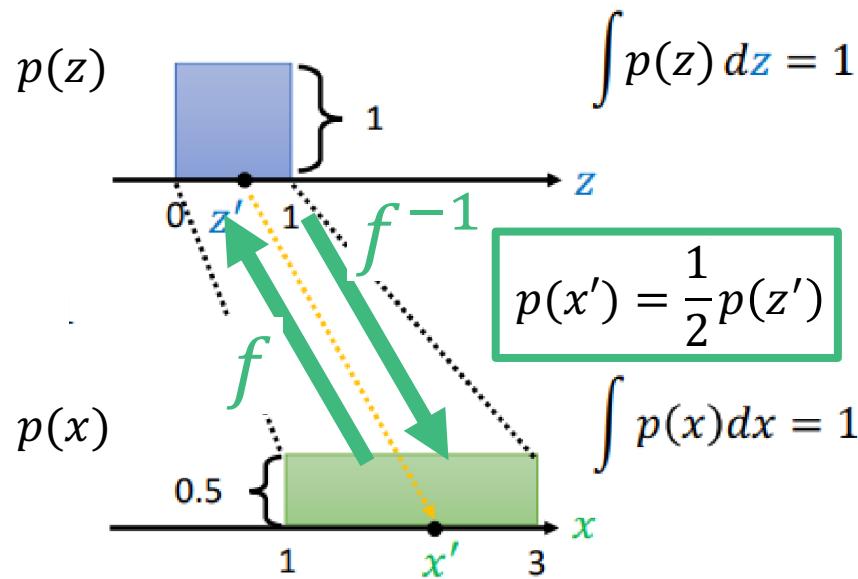


# Change of variable theorem

Define density of a random variable that is a deterministic transformation of another variable → **ensures still sums to 1!**

$$z = f(x) = \frac{x - 1}{2}$$

$$x = f^{-1}(z) = 2z + 1$$



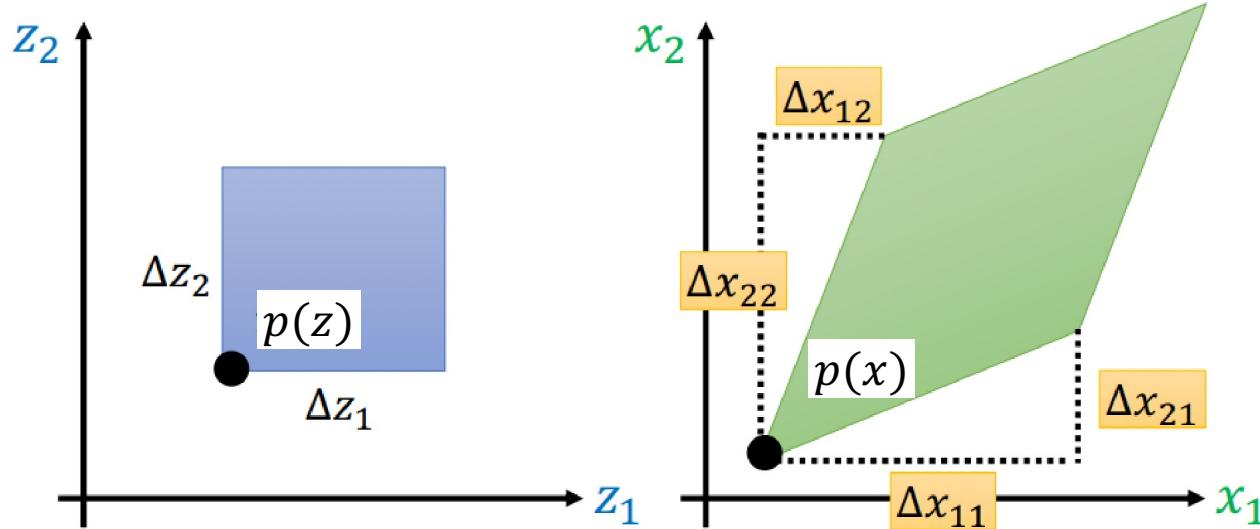
# Change of variable theorem

Given  $x \in \mathbb{R}$  and  $z \in \mathbb{R}$ , we get:

$$p_X(x) = p_Z(f(x)) \left| \frac{df(x)}{dx} \right|$$

# Change of variable theorem

Define density of a random variable that is a deterministic transformation of another variable → **ensures still sums to 1!**



## Change of variable theorem

Given  $\mathbf{x} \in \mathbb{R}^N$  and  $\mathbf{z} \in \mathbb{R}^N$  (must be same dimensionality!), we get:

$$p_X(\mathbf{x}) = p_Z(f(\mathbf{x})) \left| \det \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} \right|$$

# Change of variable theorem

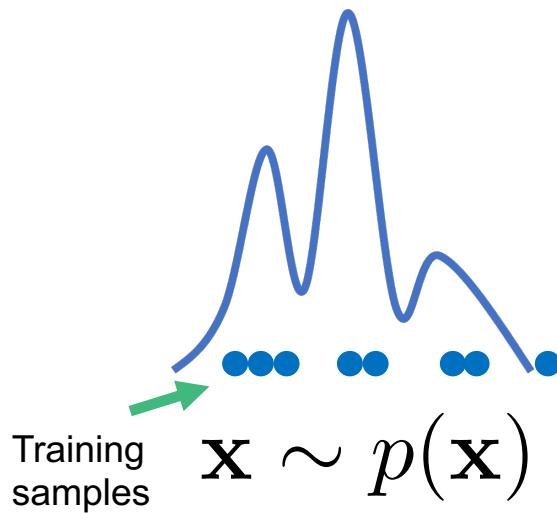
Given  $x \in \mathbb{R}^N$  and  $z \in \mathbb{R}^N$  (must be same dimensionality!), we get:

$$p_X(\mathbf{x}) = p_Z(f(\mathbf{x})) \left| \det \mathbf{J}_f \right|$$


Jacobian matrix  $\in \mathbb{R}^{N \times N}$

# Normalizing Flows: learning process

Change of variable theorem allows direct maximum-likelihood for NFs:



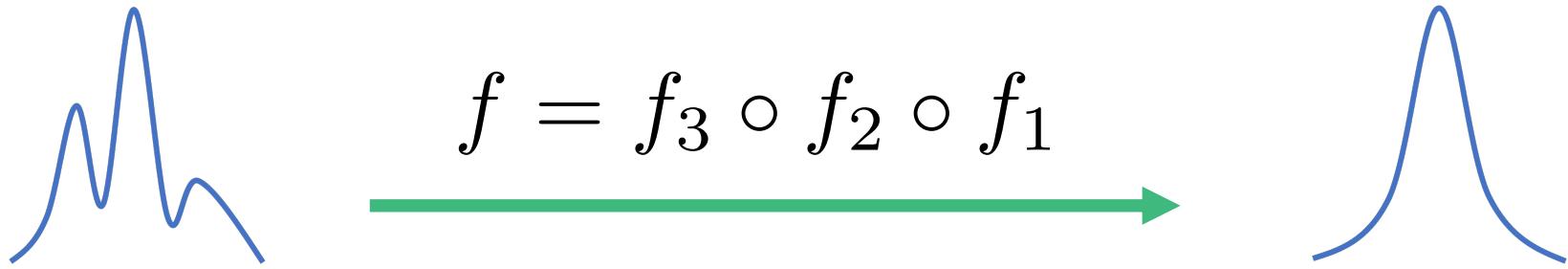
$$\operatorname{argmax}_{\theta} p_X(\mathbf{x}) =$$

$$\operatorname{argmax}_{\theta} p_Z(f_{\theta}(\mathbf{x})) |\det \mathbf{J}_{f_{\theta}}|$$

$$\operatorname{argmin}_{\theta} -\log p_Z(f_{\theta}(\mathbf{x})) - \log |\det \mathbf{J}_{f_{\theta}}|$$

# Normalizing Flows: learning process

In practice multiple transformations are required:



$$\operatorname{argmin}_{\theta} -\log p_Z(f_{\theta}(\mathbf{x})) - \sum_i \log |\det \mathbf{J}_{f_{i,\theta}}|$$

# Normalizing Flows in practice

## Training Algorithm

1. Select a batch of training samples and feed to network ( $f_\theta$ )
2. Compute the loss: 1) Evaluate the probability of each output ( $p_Z(f_\theta(x))$ ). This is easy to do as  $p_Z$  is chosen to be a simple distribution with analytical pdf (e.g., gaussian).  
2) Compute the log-determinant of the Jacobian + backprop.

## Inference Algorithm

1. Randomly sample a value from the distribution  $p_Z$
2. Feed this sample into the inverse transformation  $f_\theta^{-1}$  to obtain a sample  $x$  of  $p_X$
3. Optional: feed the sample into the network and evaluate its probability via the change of variable formula.

# Why Normalizing Flows?

**Normalizing:** *the change of variables formula gives a normalized density after applying an invertible transformation.*

**Flow:** *multiple invertible transformations can be composed with each other to create more complex invertible transformations.*

# Computational issues with NFs

Computing the log-determinant of Jacobian can be expensive!

$$\sigma(x) = \frac{1}{1 + e^{-Wx-b}}$$

Very simple MLP network


$$J_{\sigma(x)} = \left[ \frac{e^{-Wx-b}}{(1+e^{-Wx-b})^2} \odot w_1, \dots, \frac{e^{-Wx-b}}{(1+e^{-Wx-b})^2} \odot w_n \right]$$

Dense matrix, determinant roughly as expensive as inverting  $J$

# Computational issues with NFs

Computing the log-determinant of Jacobian can be expensive!

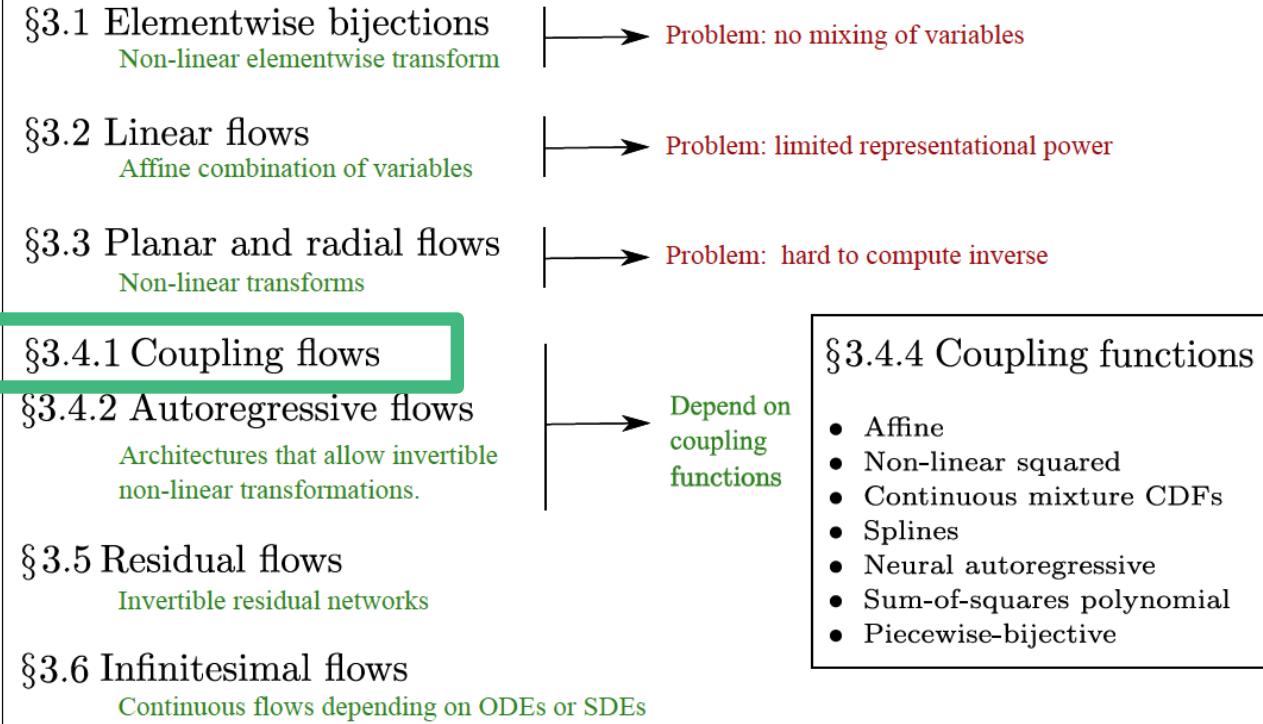
## Solution

Design **invertible** transformations whose **Jacobian has special structure**  
→ e.g., lower triangular (determinant = product of diag. elements)

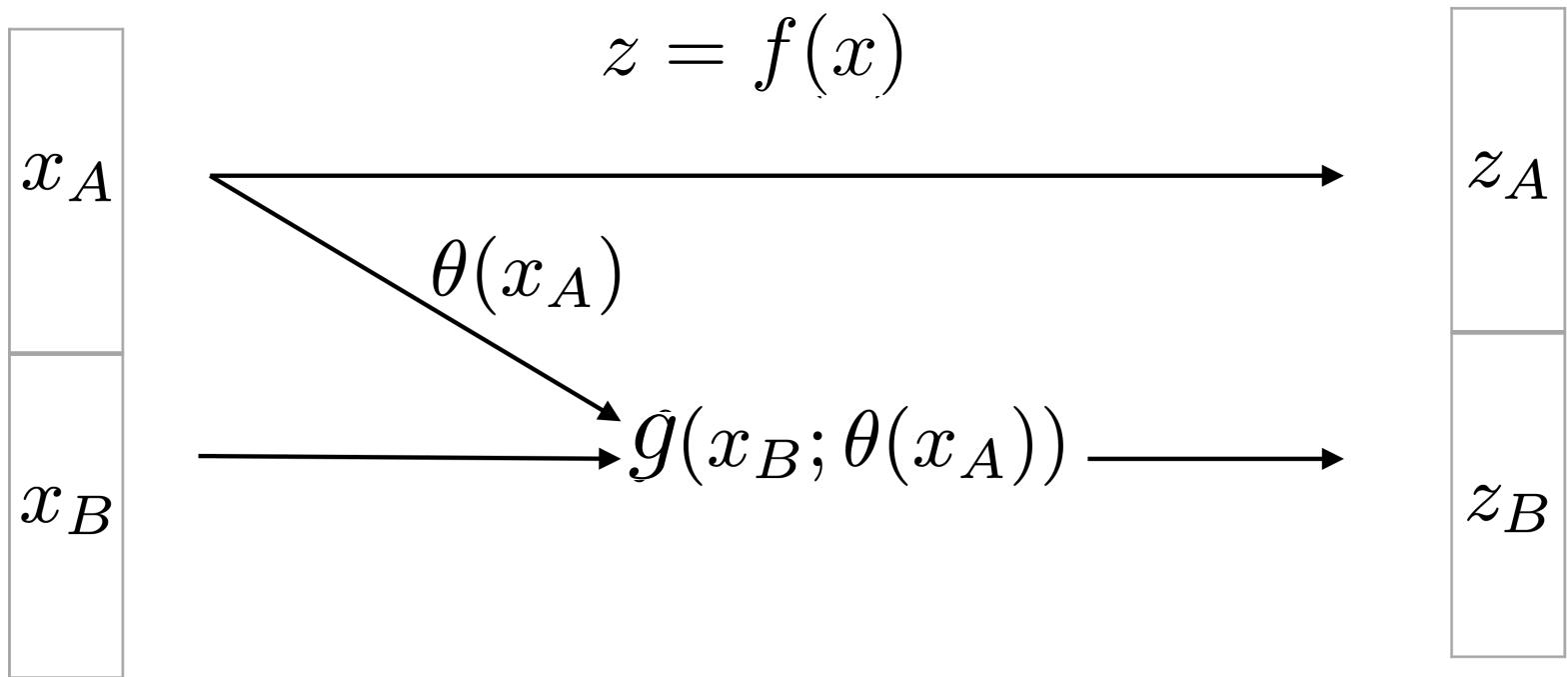


**Coupling layers/flows**

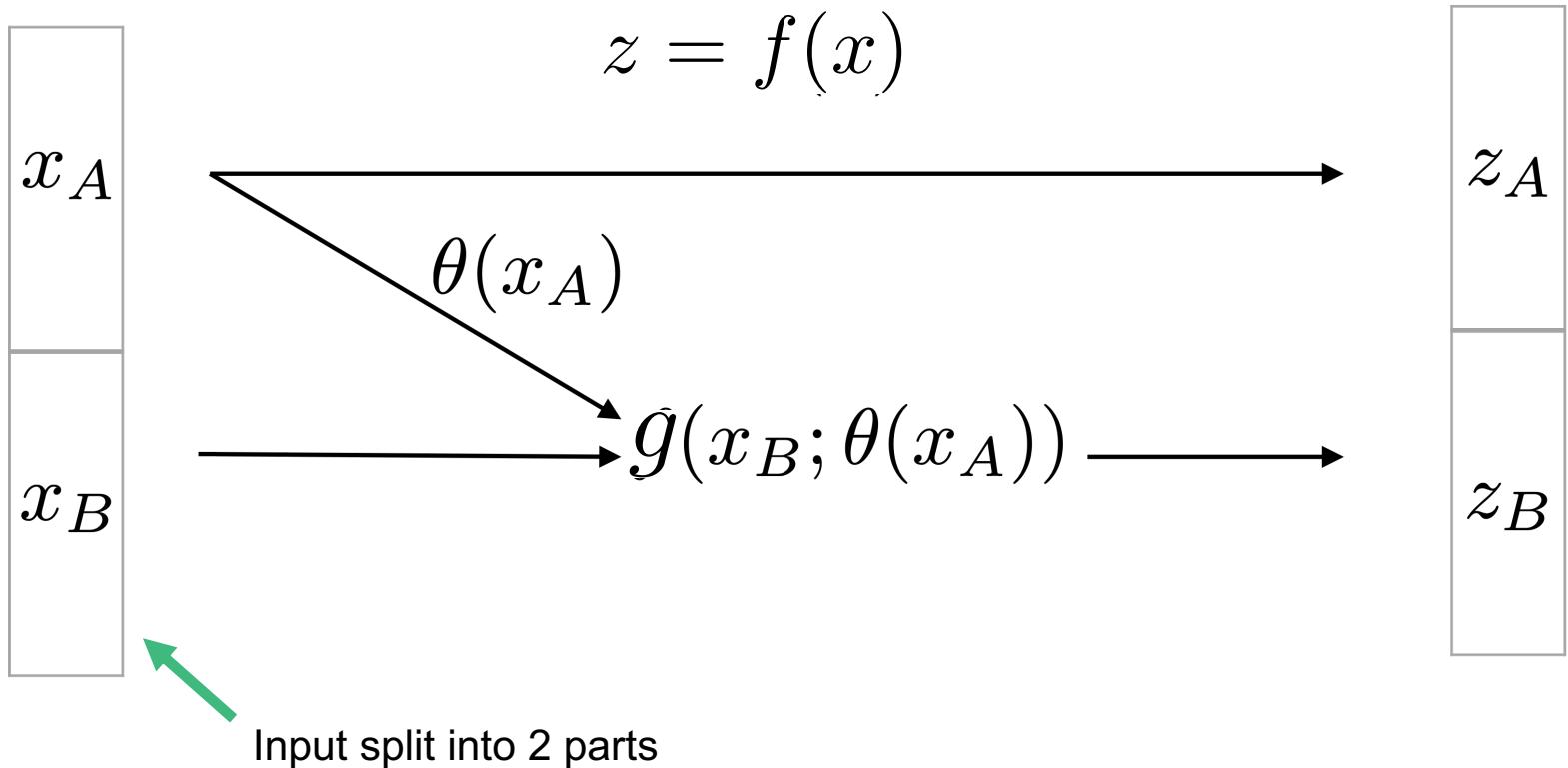
# Computational issues with NFs



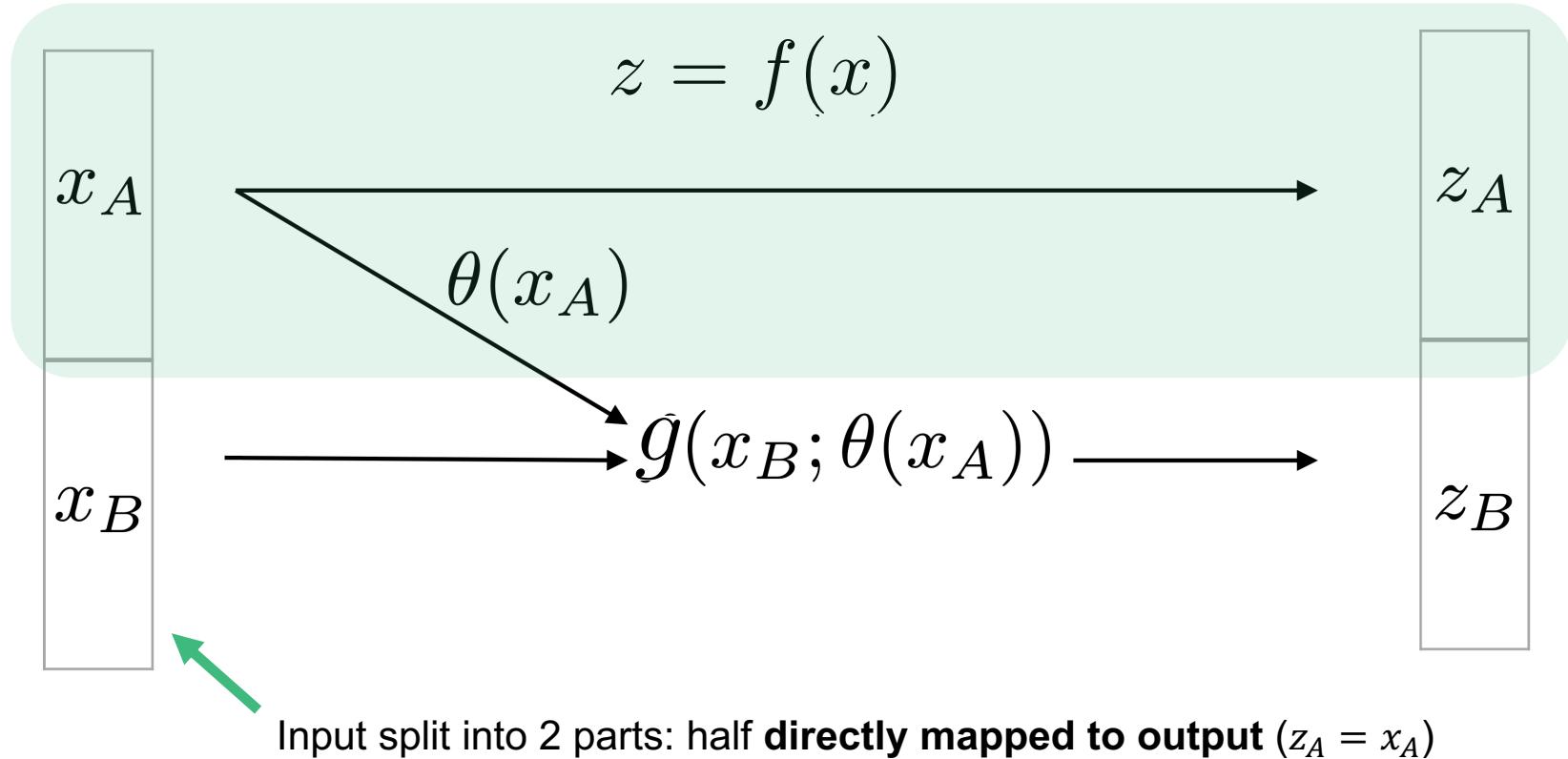
# Coupling layers/flows



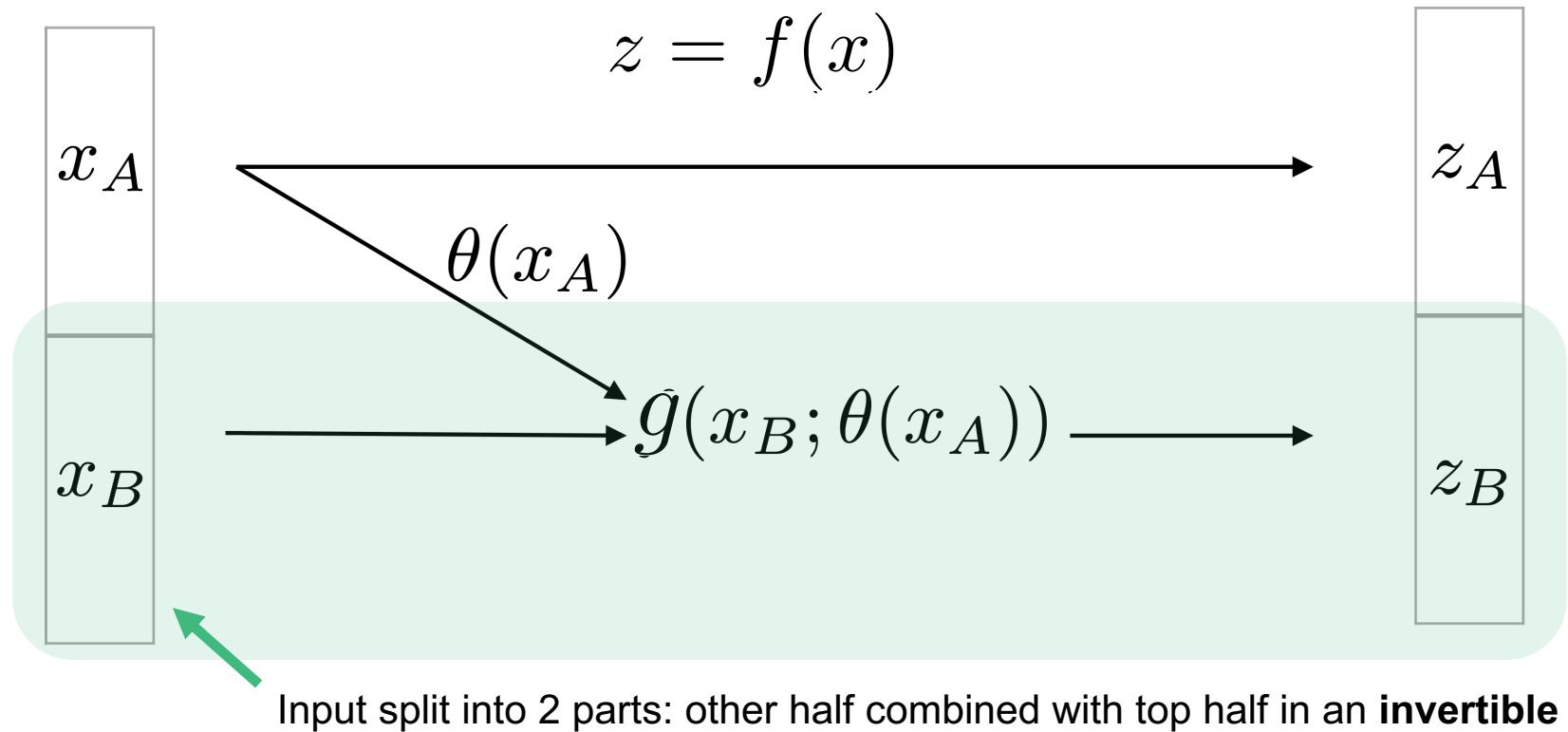
# Coupling layers/flows



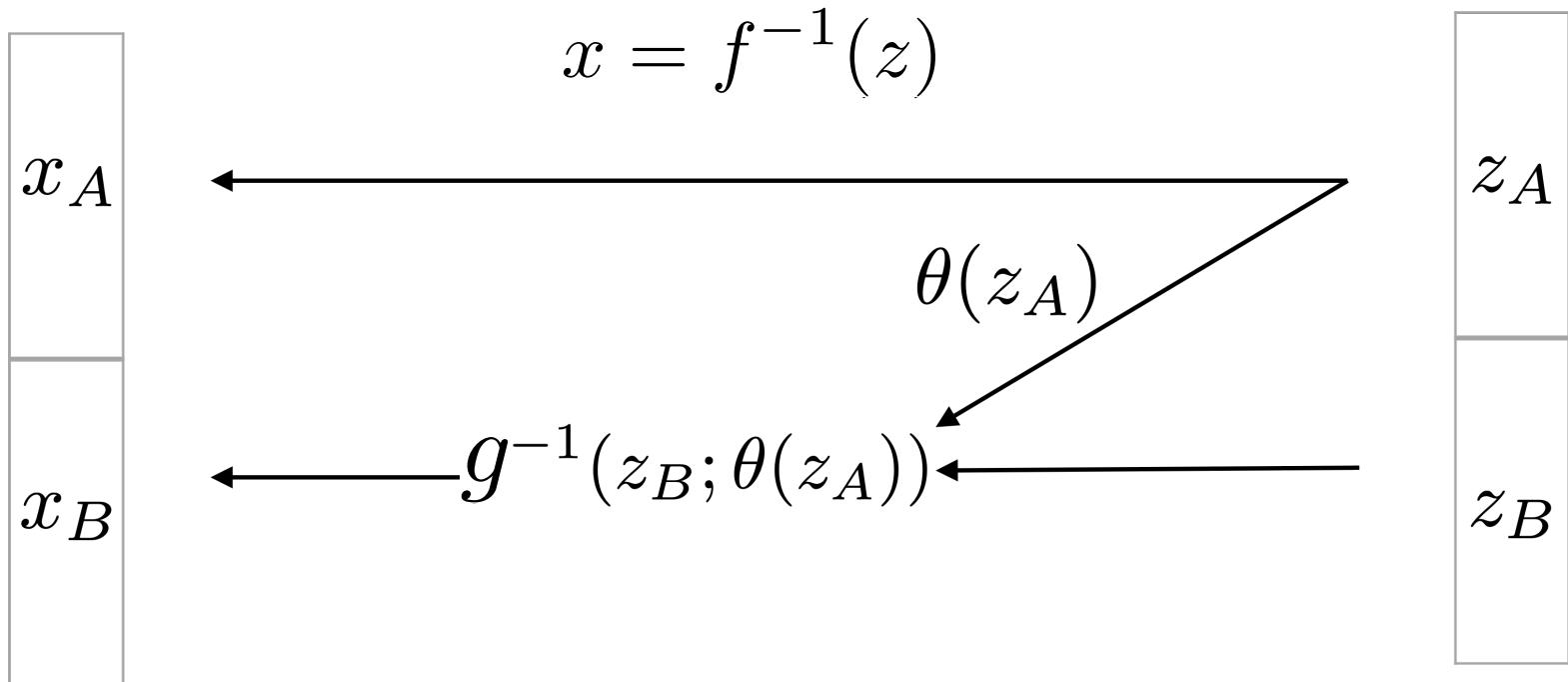
# Coupling layers/flows



# Coupling layers/flows



# Coupling layers/flows



The flow designed like this is invertible **for any choice of  $\theta$ !**

# Coupling layers/flows

Forward:

$$z_A = x_A$$

$$z_B = g(x_B; \theta(x_A))$$

Inverse:

$$x_A = z_A$$

$$x_B = g^{-1}(z_B; \theta(z_A))$$

# Coupling layers/flows

Choice of  $g$ :

- NICE (Dinh et al., 2014): Additive coupling function

$$z_B = x_B + \theta(x_A)$$

Can be NNs

- REAL NVP (Dinh et al., 2016): Affine coupling function

$$z_B = x_B \odot e^{s(x_A)} + t(x_A)$$

Scale and Transformation: Can be NNs

# Coupling layers/flows: inverse and Jacobian

Let's consider first the first scenario (i.e., additive coupling):

$$z_A = x_A$$

$$z_B = x_B + \theta(x_A)$$



$$x_A = z_A$$

$$x_B = z_B - \theta(x_A)$$

$$J_f = \begin{bmatrix} \partial z_A / \partial x_A & \partial z_A / \partial x_B \\ \partial z_B / \partial x_A & \partial z_B / \partial x_B \end{bmatrix} = \begin{bmatrix} I & 0 \\ \partial \theta(x_A) / \partial x_A & I \end{bmatrix}$$

$$\det J_f = 1$$

Irrelevant..

## Coupling layers/flows: inverse and Jacobian

Let's consider first the first scenario (i.e., additive coupling):

$$\operatorname{argmin}_{\theta} - \log p_Z(f_{\theta}(\mathbf{x})) - \sum_i \log |\det \mathbf{J}_{f_i, \theta}|$$

~~$\log |\det \mathbf{J}_{f_i, \theta}| = 0$~~

# Coupling layers/flows: inverse and Jacobian

Let's consider first the second scenario (i.e., affine coupling):

$$\begin{aligned} z_A &= x_A \\ z_B &= x_B \odot e^{s(x_A)} + t(x_A) \end{aligned} \quad \xrightarrow{\text{green arrow}} \quad \begin{aligned} x_A &= z_A \\ x_B &= (z_B - t(x_A)) \odot e^{-s(x_A)} \end{aligned}$$

$$J_f = \begin{bmatrix} \partial z_A / \partial x_A & \partial z_A / \partial x_B \\ \partial z_B / \partial x_A & \partial z_B / \partial x_B \end{bmatrix} = \begin{bmatrix} I & 0 \\ \partial z_B / \partial x_A & \text{diag}\{e^{s(x_A)}\} \end{bmatrix}$$

$$\det J_f = \prod_i e^{s(x_A)_i}$$

Irrelevant..

# Coupling layers/flows

---

Choice of  $\theta$ :

- NICE (Dinh et al., 2014): MLP with ReLU
- REAL NVP (Dinh et al., 2016): Convolution residual layer
- Glow (Kingma et al., 2018): 3-layer CNN with ReLU
- ...

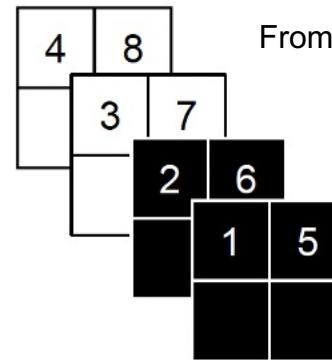
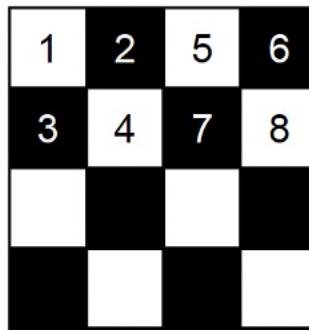
# Split functions

What is the most effective way to partition  $x$ ?

- Separating the input in half and passing the first half untouched leads to **poor expressivity!**
- Use some sort of **shuffling** before separating → better...
- More structured ways:
  - REAL NVP (Dinh et al., 2016): Masking
  - Glow (Kingma et al., 2018): 1x1 Conv.

# Split functions

What is the most effective way to partition  $x$ ?



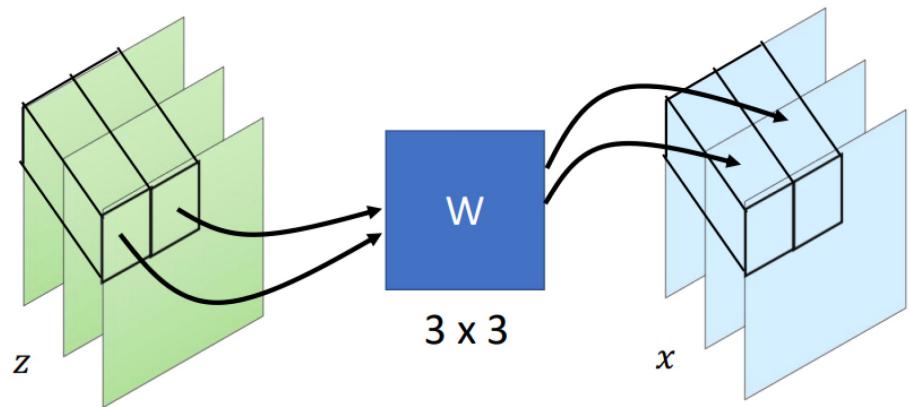
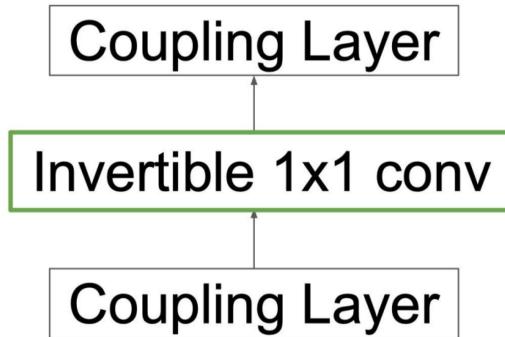
From REAL NVP (Dinh et al., 2016)

Figure 3: Masking schemes for affine coupling layers. On the left, a spatial checkerboard pattern mask. On the right, a channel-wise masking. The squeezing operation reduces the  $4 \times 4 \times 1$  tensor (on the left) into a  $2 \times 2 \times 4$  tensor (on the right). Before the squeezing operation, a checkerboard pattern is used for coupling layers while a channel-wise masking pattern is used afterward.

# Split functions

What is the most effective way to partition  $x$ ?

From Glow (Kingma et al., 2018)



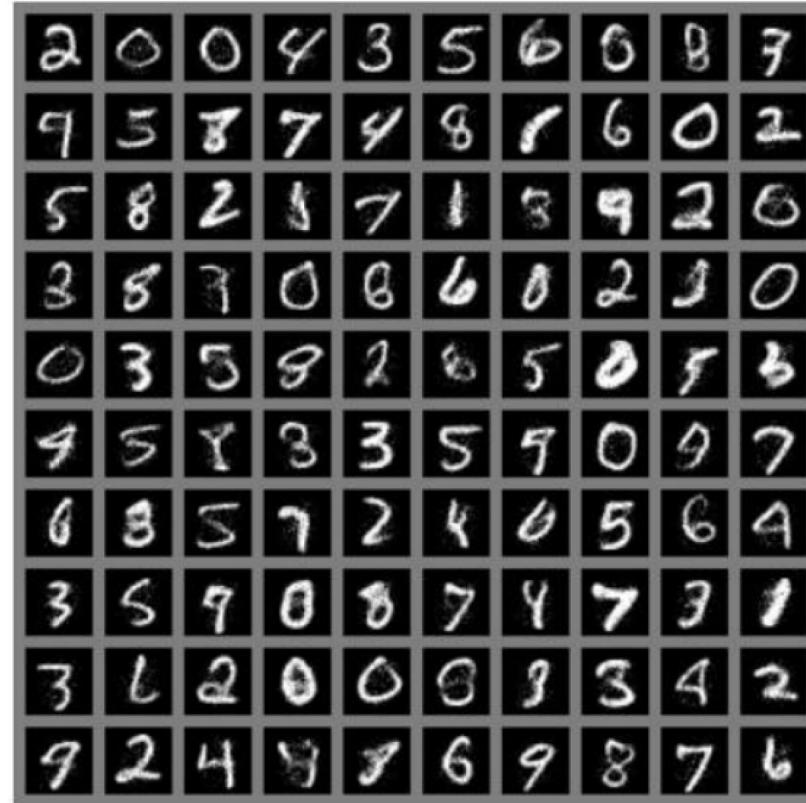
$W$  can shuffle the channels.

If  $W$  is invertible, it is easy to compute  $W^{-1}$ .

$$\begin{matrix} 3 \\ 1 \\ 2 \end{matrix} = \begin{matrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix} \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

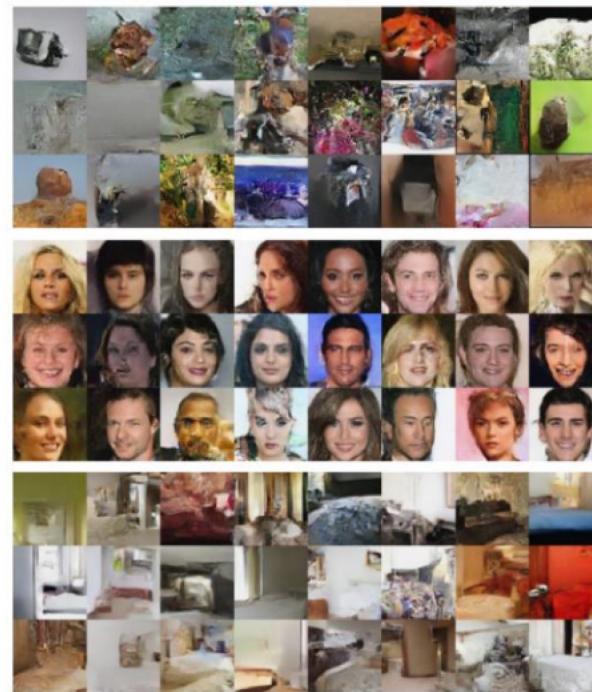
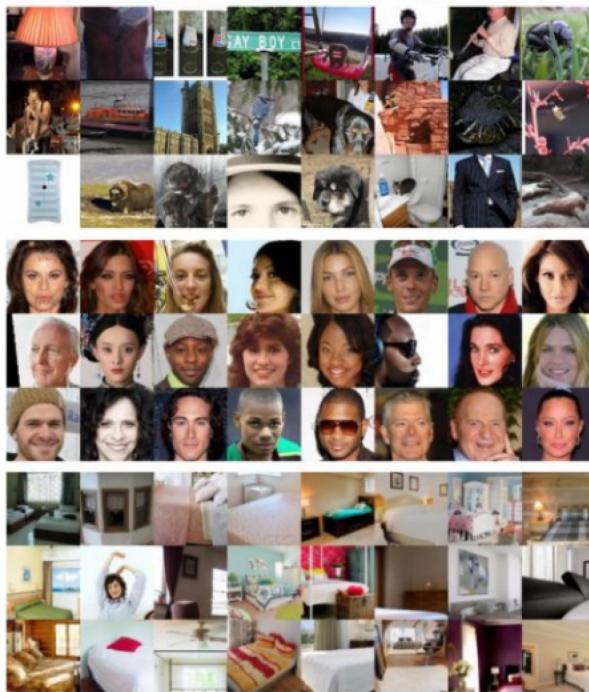
# NFs in action

---



MNIST from NICE

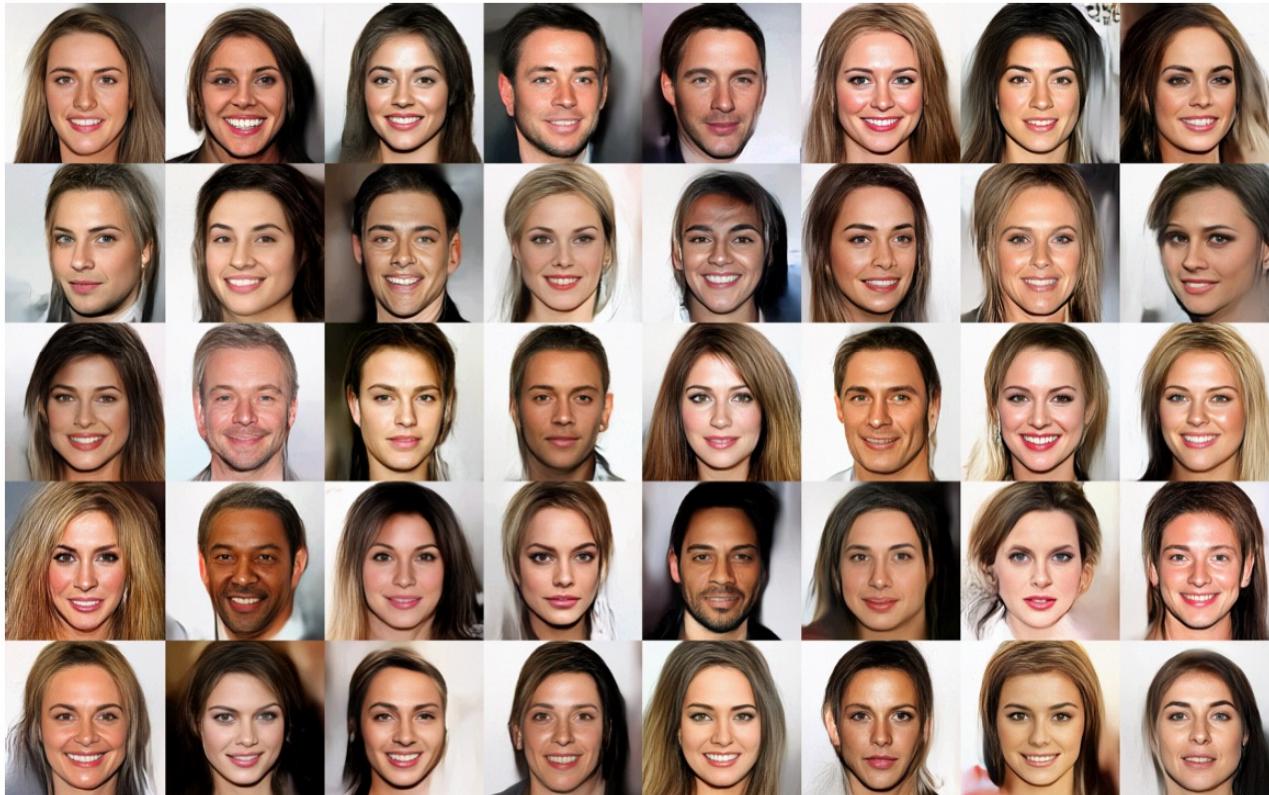
# NFs in action



## Generation from RealNVP

# NFs in action

---



Generation from Glow