Tractable density models- Normalizing Flows

Generative and Deep Learning (GDL)

Davide Bacciu (davide.bacciu@unipi.it)

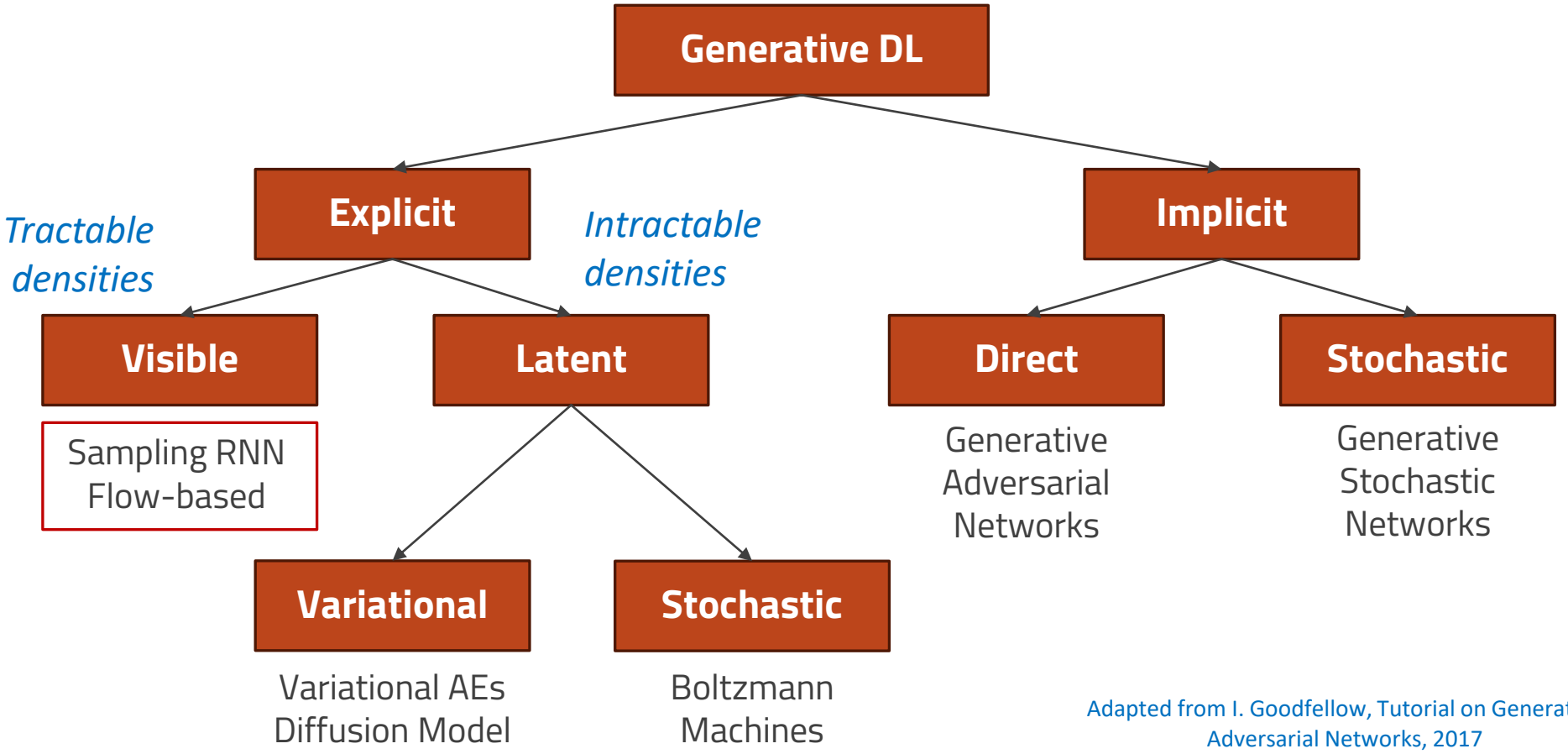
UNIVERSITÀ DI PISA

Lecture(s) outline

Generative models with explicit likelihood

- Autoregressive learning with fully visible information
- Explicitly likelihood by change of variables
- Normalizing flows
  - Coupling flows
  - Masking & squeezing
  - Invertible convolutions
  - Autoregressive flows
- Residual and continuous normalizing flows

A Taxonomy



Adapted from I. Goodfellow, Tutorial on Generative Adversarial Networks, 2017

Explicit likelihood models

Learning with Fully Visible Information

If all information is fully visible the joint distribution can be computed from the chain rule factorization

Bayesian Networks → $P(\boldsymbol{x}) = \prod_{i}^{N} P(x_i | x_1, \dots, x_{i-1})$

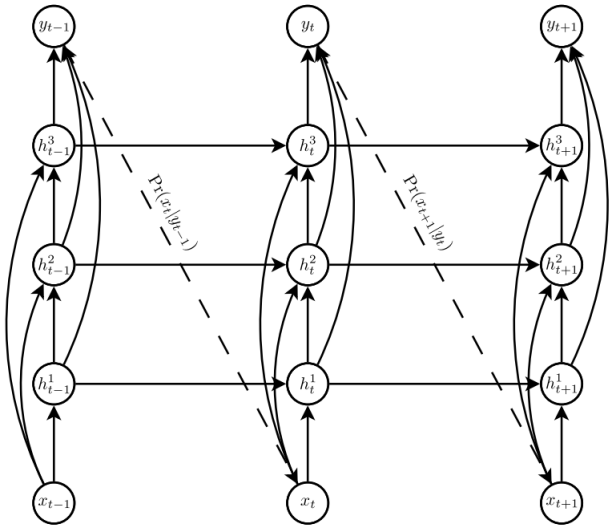
Probability of a pixel having a certain intensity value, given the known intensity of its predecessor

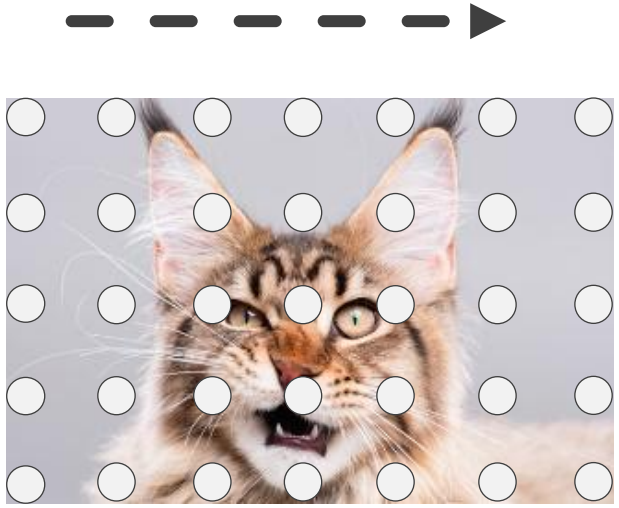
Need to be able to define a sensible ordering for the chain rule

Conditional distribution difficult to compute

Approximating the Conditional Probability

If all information is fully visible the joint distribution can be computed from the chain rule factorization

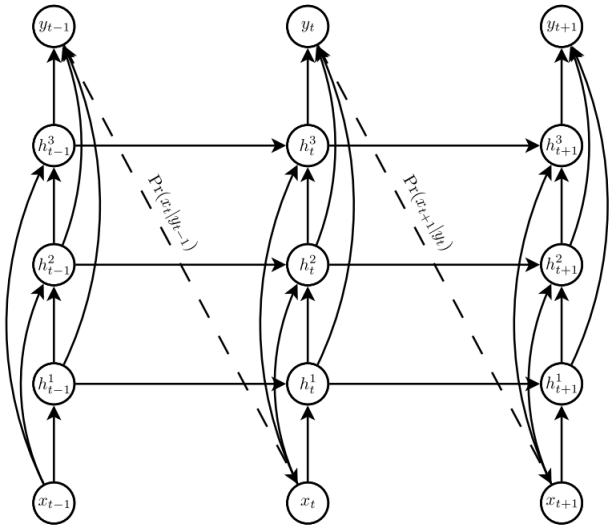


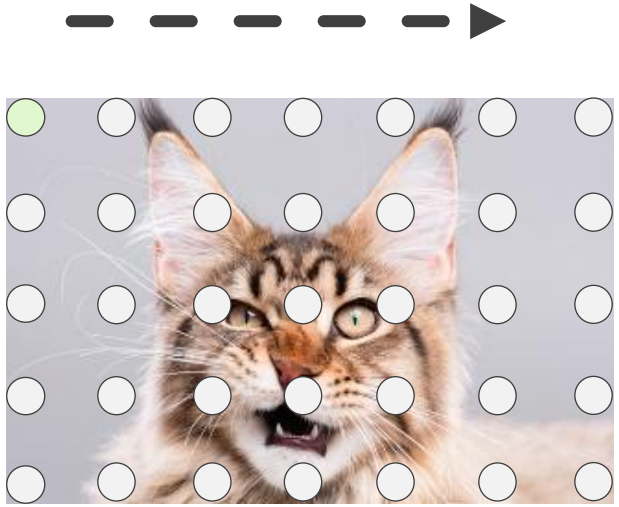


Scan the image according to a schedule and encode the dependency from previous pixels in the states of an RNN

Approximating the Conditional Probability

If all information is fully visible the joint distribution can be computed from the chain rule factorization

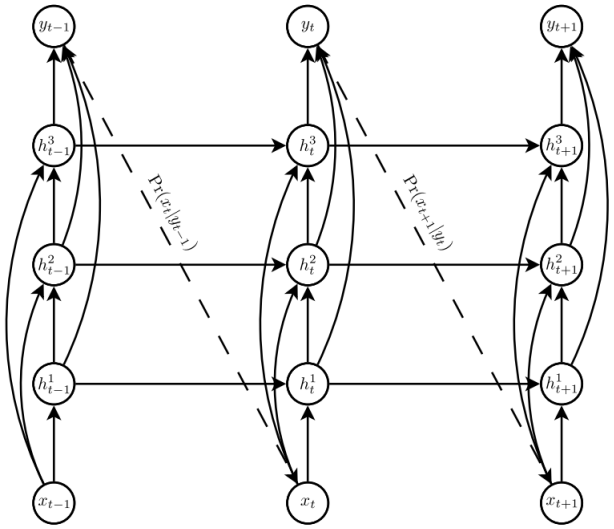


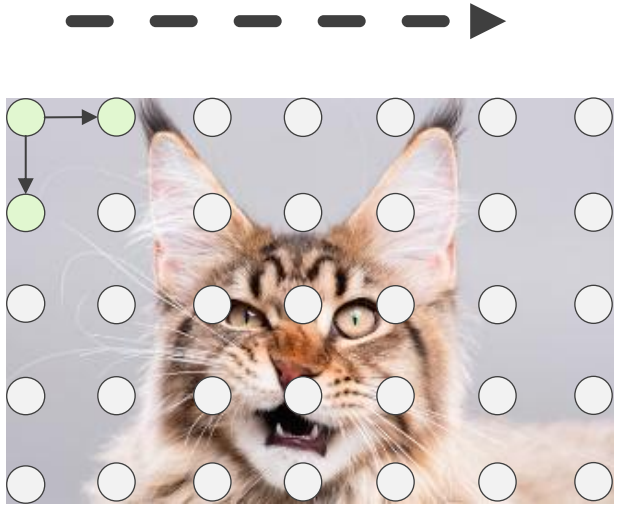


Scan the image according to a schedule and encode the dependency from previous pixels in the states of an RNN

Approximating the Conditional Probability

If all information is fully visible the joint distribution can be computed from the chain rule factorization

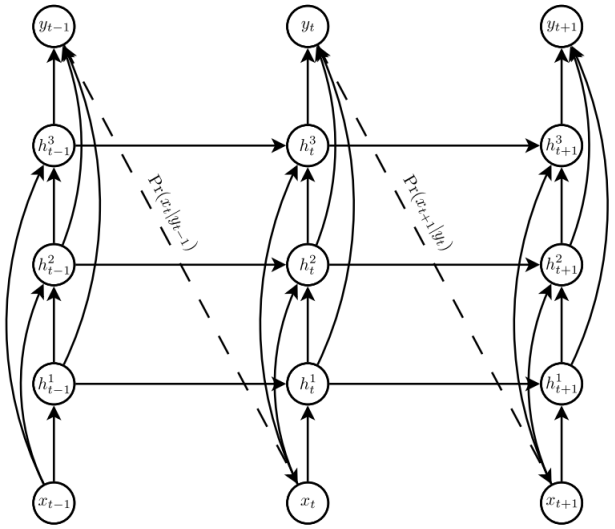


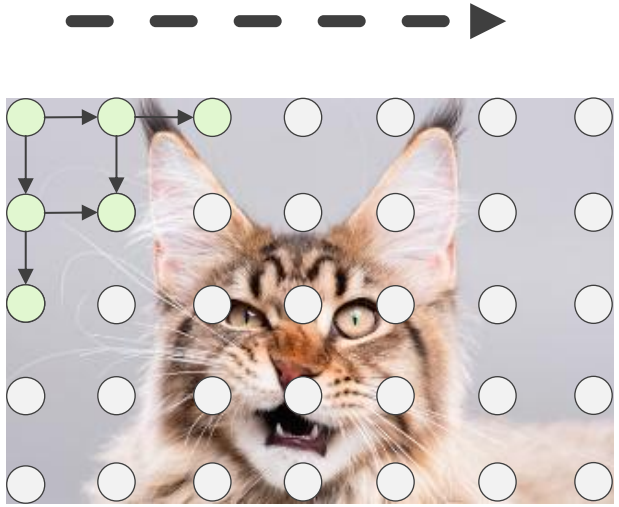


Scan the image according to a schedule and encode the dependency from previous pixels in the states of an RNN

Approximating the Conditional Probability

If all information is fully visible the joint distribution can be computed from the chain rule factorization

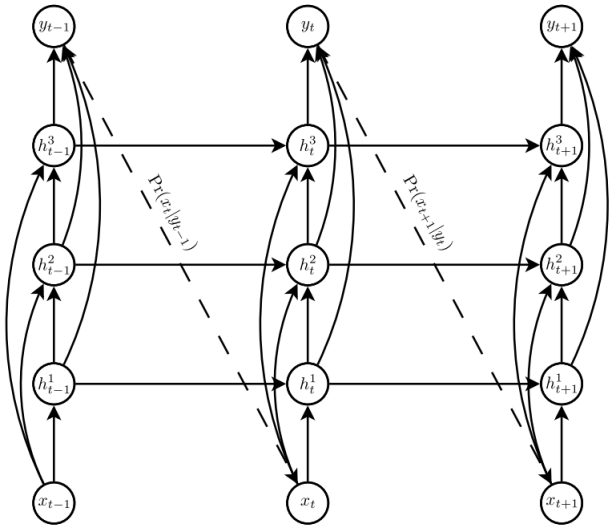


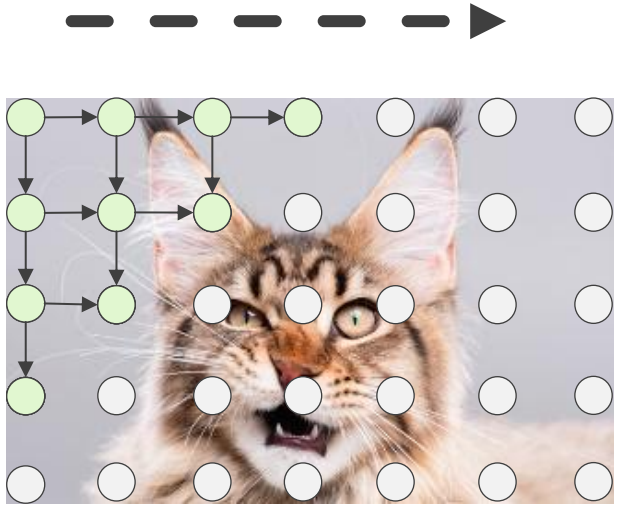


Scan the image according to a schedule and encode the dependency from previous pixels in the states of an RNN

Approximating the Conditional Probability

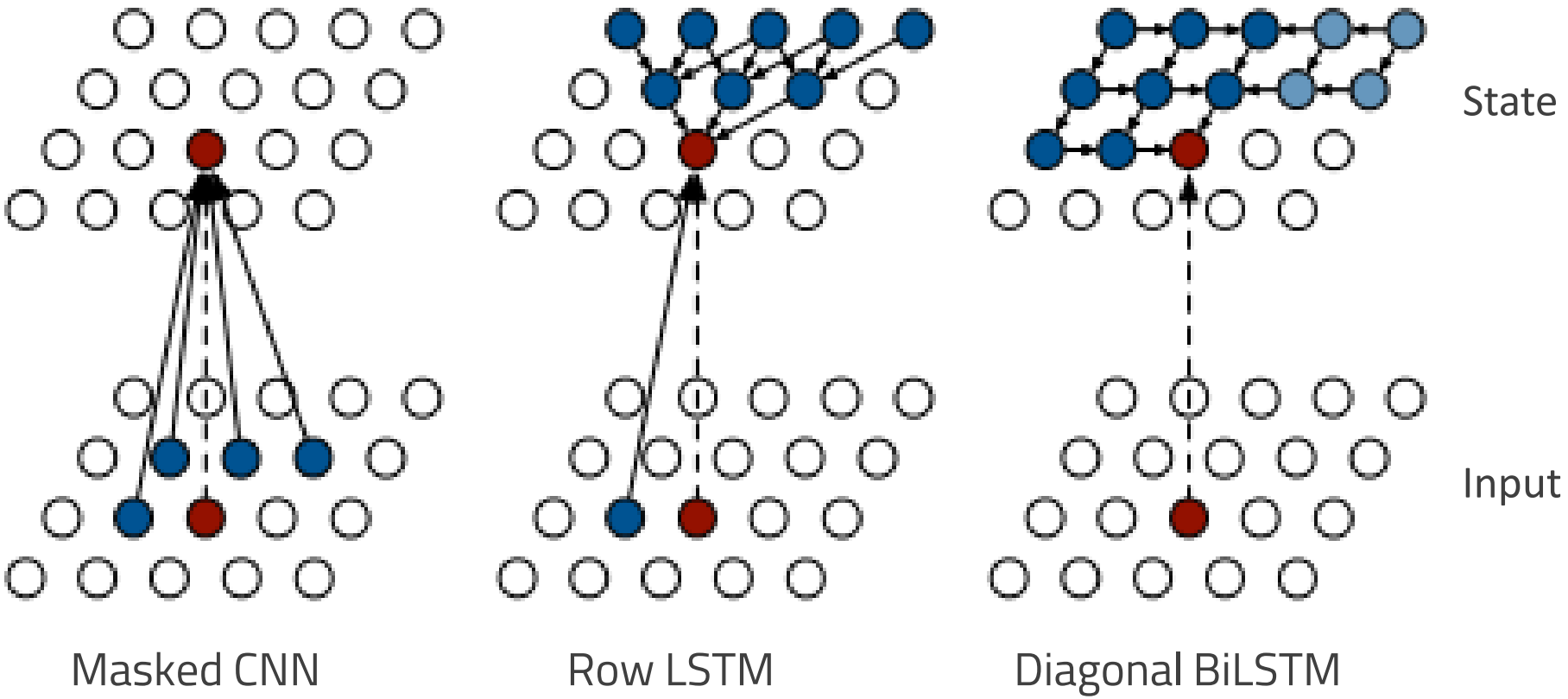
If all information is fully visible the joint distribution can be computed from the chain rule factorization





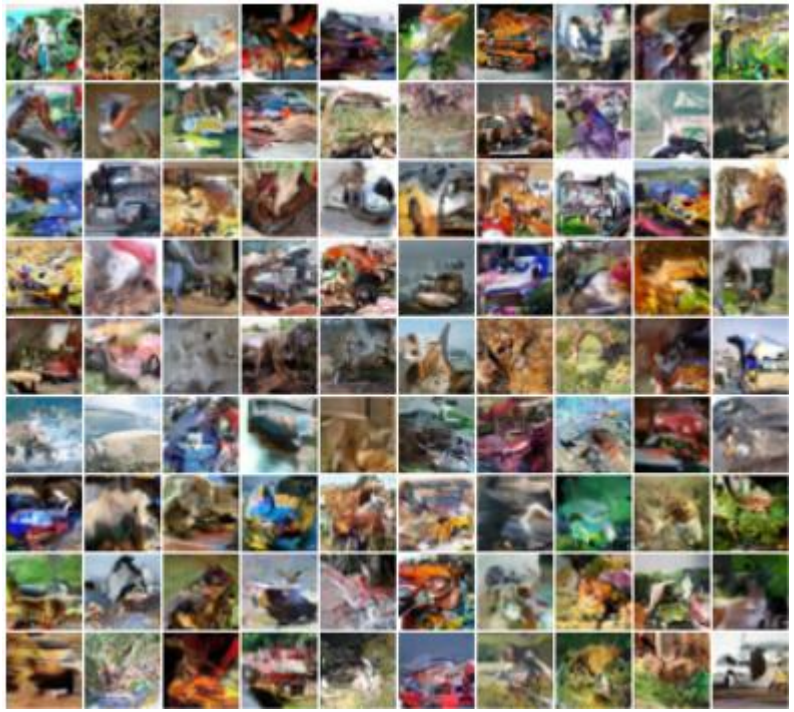
Scan the image according to a schedule and encode the dependency from previous pixels in the states of an RNN

Generating Images Pixel by Pixel

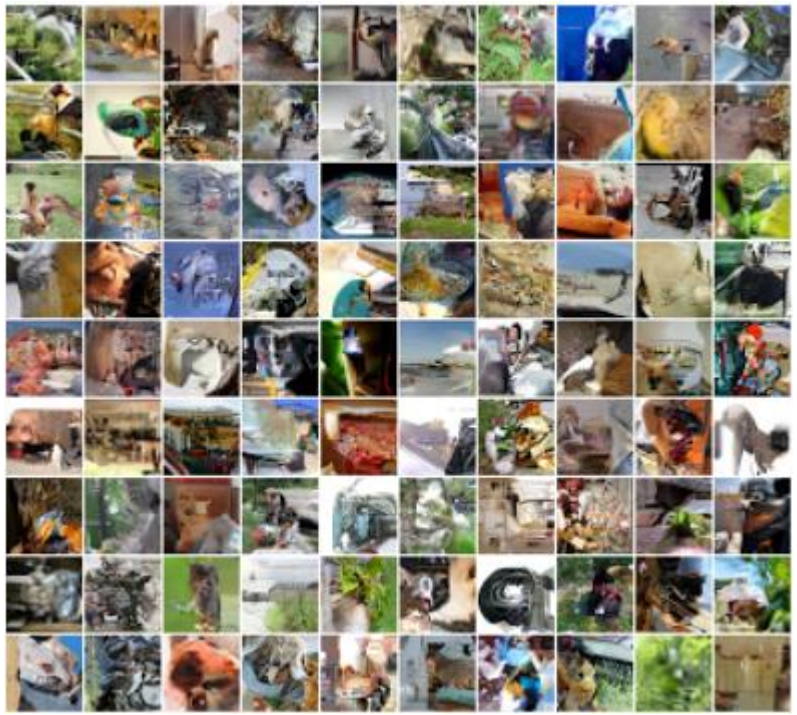


A. van der Oord et al., Pixel Recurrent Neural Networks, 2016

Generating Images Pixel by Pixel - Results



32x32 CIFAR-10



32x32 ImageNet

A. van der Oord et al., Pixel Recurrent Neural Networks, 2016

Summary on autoregressive models

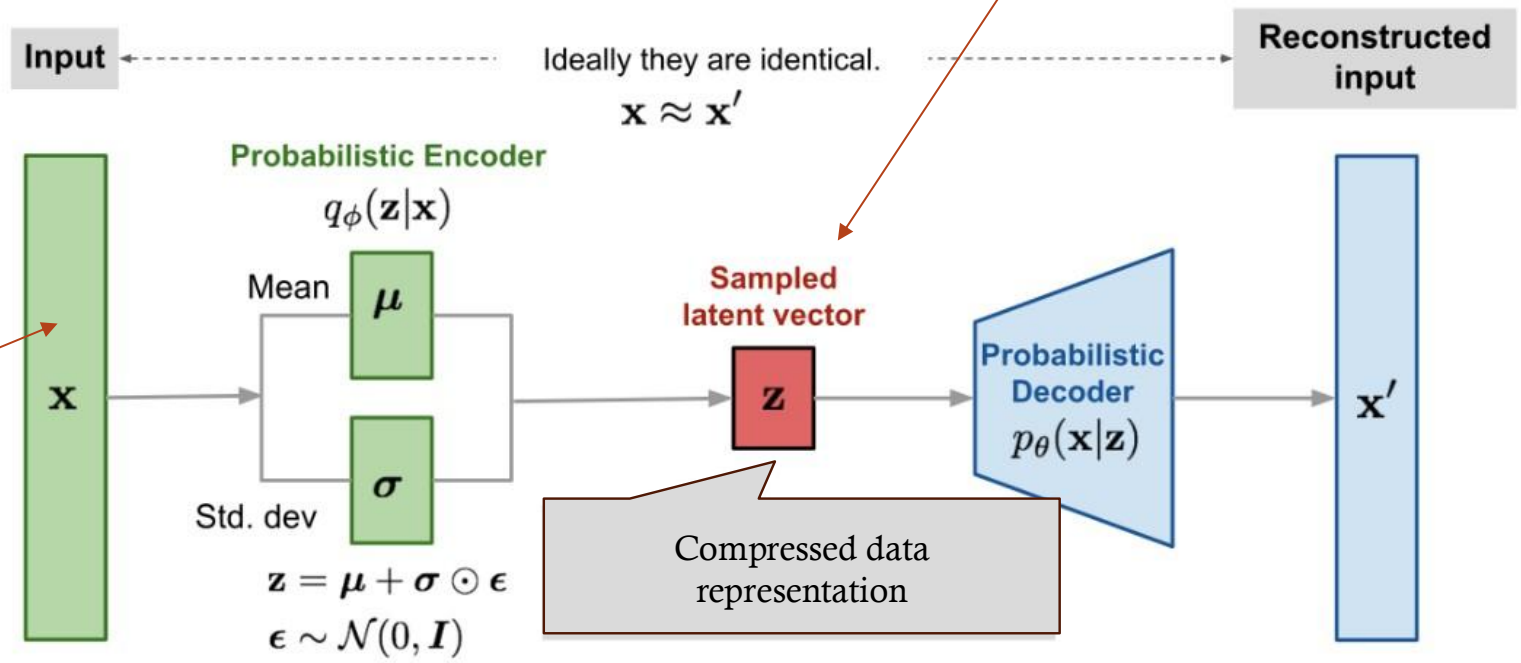
- Autoregressive models are easy to understand and design
  - Their left-to-right factorization is a natural fit for modeling text
  - They are at the core of modern LLMs (using transformers in place of RNNs/CNNs)
- Computing the likelihood is easy in general and maximizing it is easy for text (made of discrete tokens)
- Variable-length generations can be sampled one token at a time
- They are very slow to sample (sequential)
- No natural way to get meaningful latent representations
- Sub-optimal sample quality beyond text

Intuition and Motivation

Remember me? The VAE

Requires to capture all relevant variations in a low-dimensional embedding **z**

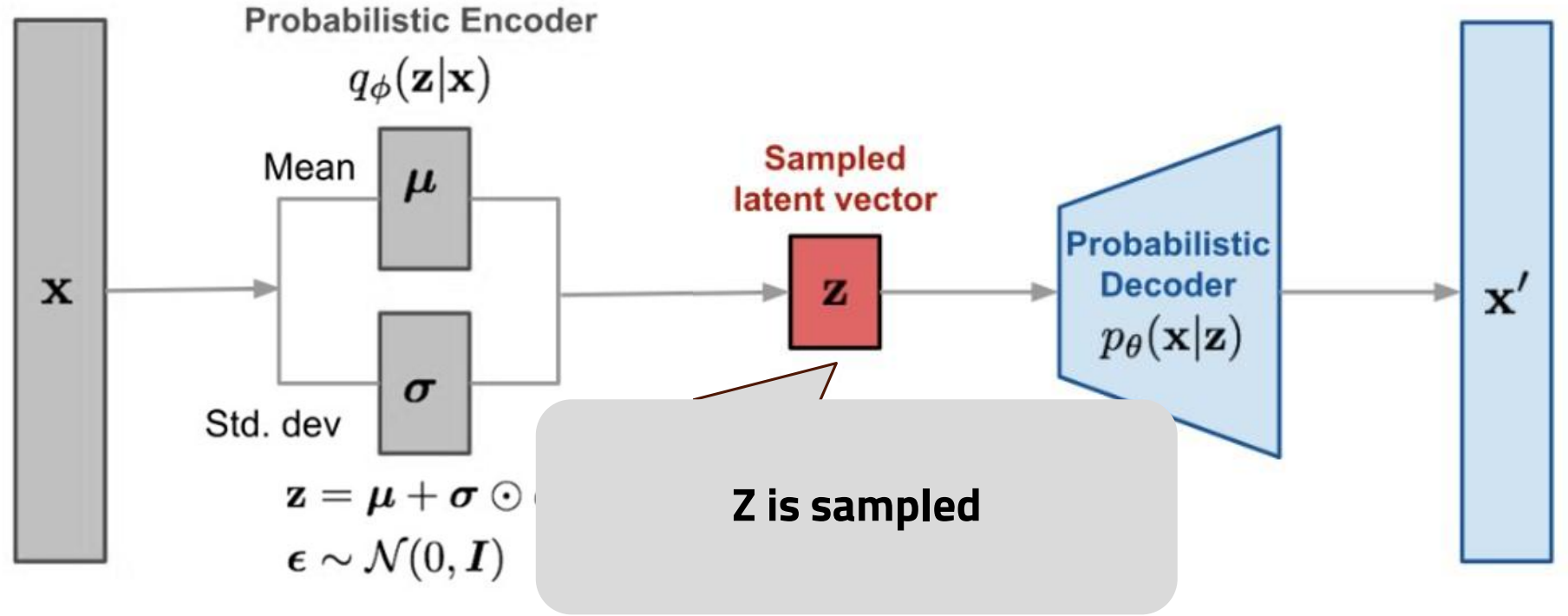
Needs separate encoder/decoder networks



$$\mathbb{E}_{X\sim D}[\mathbb{E}_{z\sim Q}[\log P(x|z)] - KL(Q(z|x,\phi)||P(z))]$$

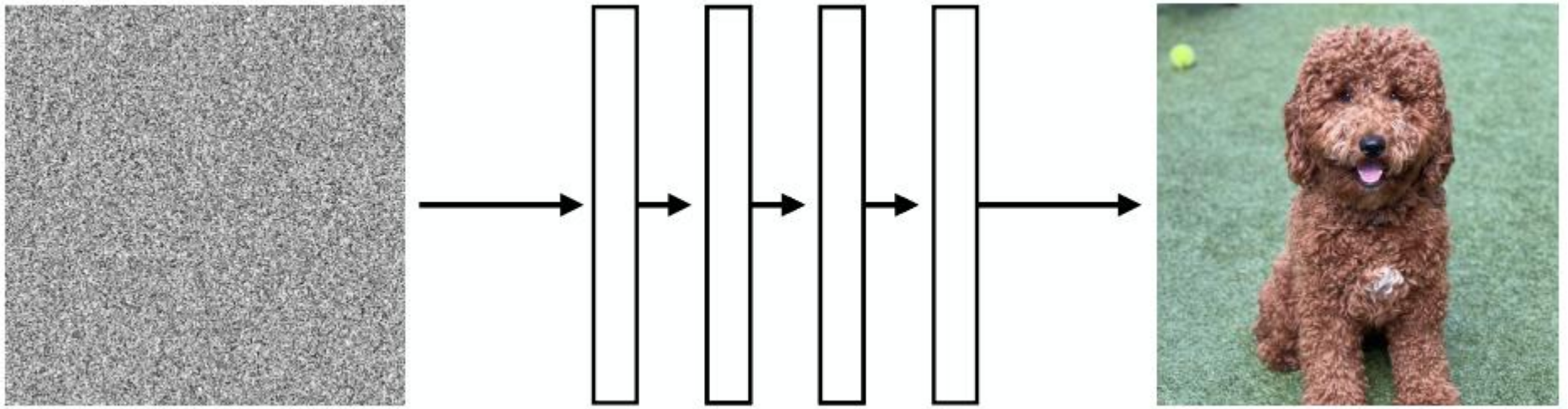
Cannot directly estimate $P_\theta(\boldsymbol{x})$

Can we get rid of the encoder?



Invertible Transformations

The generative direction



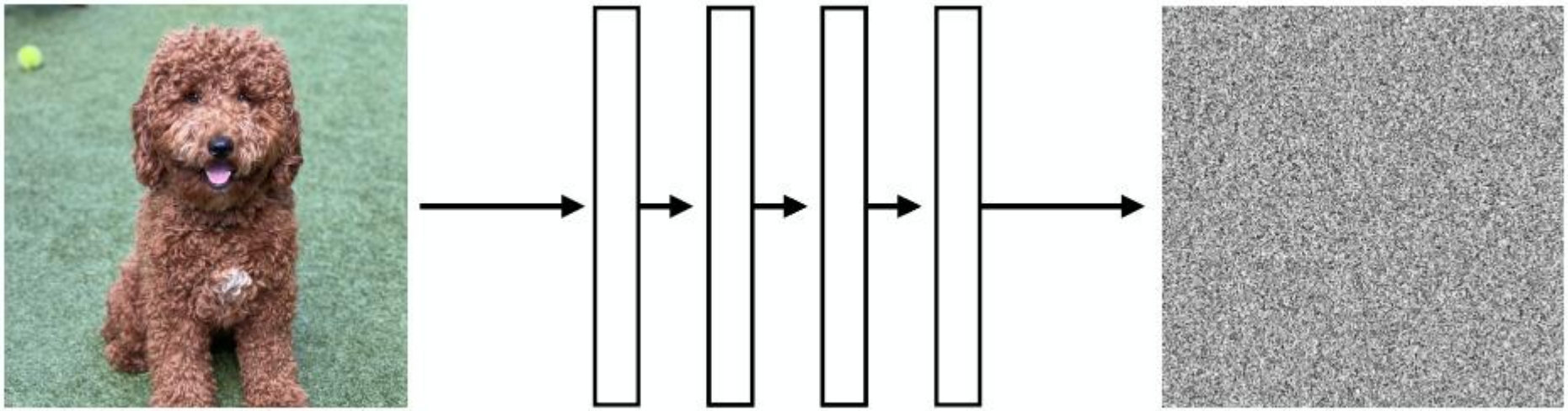
$\boldsymbol{z}$ sampled from a simple distribution

$\boldsymbol{x} = g(\boldsymbol{z})$

Reconstructed as a sample from the data distribution $\boldsymbol{x}$

Invertible Transformations

The normalizing direction



$\boldsymbol{x}$ data sample

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

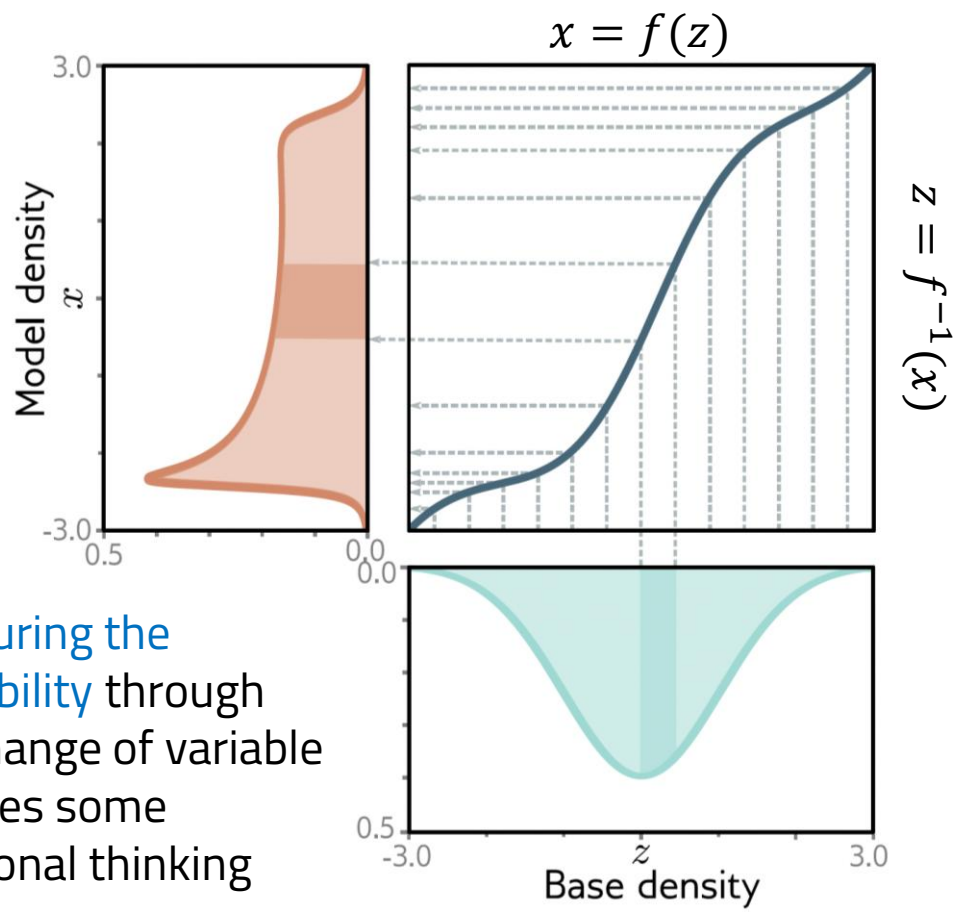
Transformed in noise $\boldsymbol{z}$

Normalizing Flow (NF) – The Intuition

- Learn a probabilistic model by transforming a simple distribution into the complex data generating distribution using a deep network
  - Easy to sample and evaluate the probability
  - Requires a specialized architecture where each layer must be invertible

Change of variable

Probabilistic Change of Variable



Measuring the probability through the change of variable requires some additional thinking

- Take a tractable base distribution $P(z)$ over latent variable z and a model density $P(x)$ over data x
- Apply a change of variable function (possibly learned with parameters θ)

$$x = f(z; \theta)$$

- In addition, we are going to require that f is invertible

$$z = f^{-1}(z; \theta)$$

Linear 1D Change of Variable

NF define complex densities by transforming a base one by invertible mappings (bijections)

- Simplest case in 1D is a univariate Gaussian base density

$$z \sim \mathcal{N}(0,1)$$

- Simplest change of variable (forward) by linear transformation

$$x = f(z; \mu, \sigma) = \mu + z\sigma$$

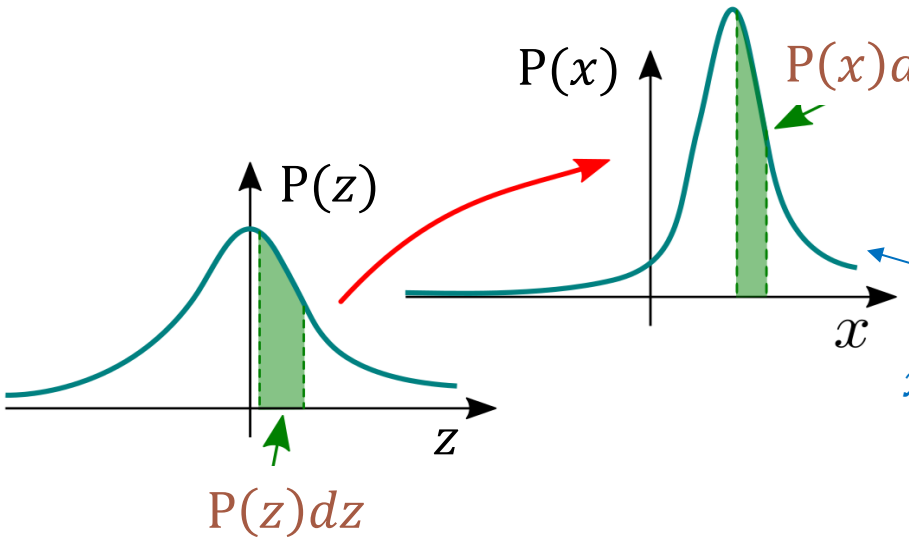
- Inverse then (under $\sigma \neq 0$)

$$z = f^{-1}(x; \mu, \sigma) = (x - \mu)/\sigma$$

- With $\mathrm{P}(z)$ known we want to find $\mathrm{P}(\mathrm{x})$

Linear 1D – Mass conservation

The volume may change but the density must be preserved



The necessary condition for this is

$$\mathrm{P}(z)dz = P(x)dx$$

The probability of data x under the transformed distribution is

$$P(x) = P(z)\left|\frac{dx}{dz}\right|^{-1} = \frac{P(z)}{\sigma}$$

$x \sim \mathcal{N}(\mu, \sigma^2)$

$\frac{x-\mu}{\sigma}$

Change of volume

$x = \mu + \sigma z$

Linear 1D – Iterated forward pass

$$P(x) = P(z)\left|\frac{dx}{dz}\right|^{-1}$$

Forward transformation equation

- Sample x through 2 mappings (transformations)

$$z_0 \sim P(z) \quad z_1 = f_1(z_0) \quad x = f_2(z_1)$$

- Density obtained by composing forward transformations

$$P(x) = P(z_0)\left|\frac{dz^1}{dz^0}\right|^{-1}\left|\frac{dx}{dz^1}\right|^{-1}$$

Linear 1D – Inverse Flow

- We may be interested in estimating the density of a given input sample x
- Requires building the inverse flow ($g = f^{-1}$)

$$z_1 = f_2^{-1}(x) = g_2(x) \qquad z_0 = f_1^{-1}(z_1) = g_1(z_1)$$

- And computing the density accordingly

$$P(x) = \left|\frac{dz^1}{dx}\right| \left|\frac{dz^0}{dz^1}\right| P(z_0)$$

Multidimensional flow

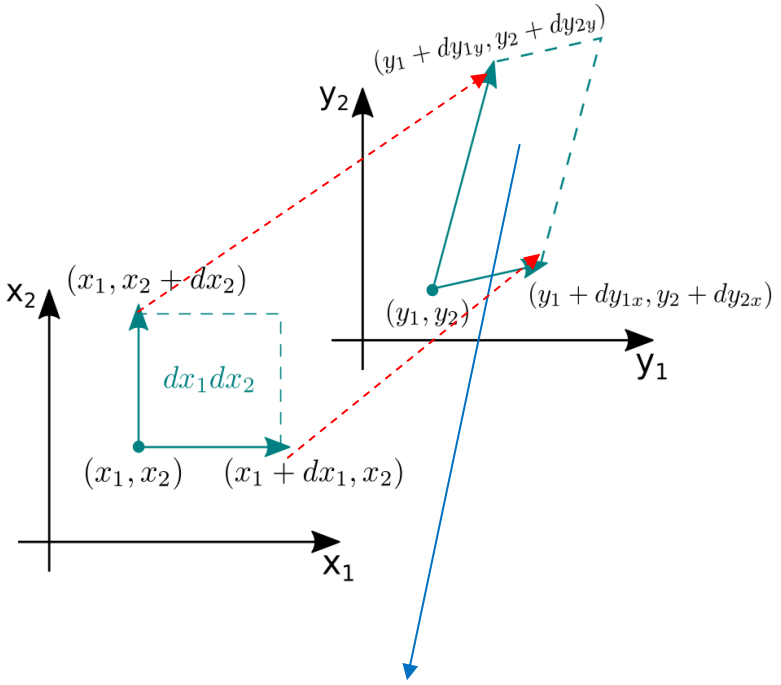
- Extend the approach to multi-dimensional case
  - $\boldsymbol{x}, \boldsymbol{z}$ vectorial RVs with density $\mathrm{P}(\boldsymbol{z})$ and $\mathrm{P}(\boldsymbol{x})$
  - Flow $\mathrm{f}(\boldsymbol{z})$ invertible and differentiable (closed under composition)
- Transformation $\mathbf{x} = \mathrm{f}(\boldsymbol{z})$ leads to the probability change

$$P(\boldsymbol{x}) = P(\boldsymbol{z}) \left| det \frac{\partial f(\boldsymbol{z})}{\partial \boldsymbol{z}} \right|^{-1}$$

Determinant

Jacobian

Provides information on the rate of change of the volume affected by the f transformation



Area of this field can be computed with vector calculus and turns out to be the Jacobian determinant

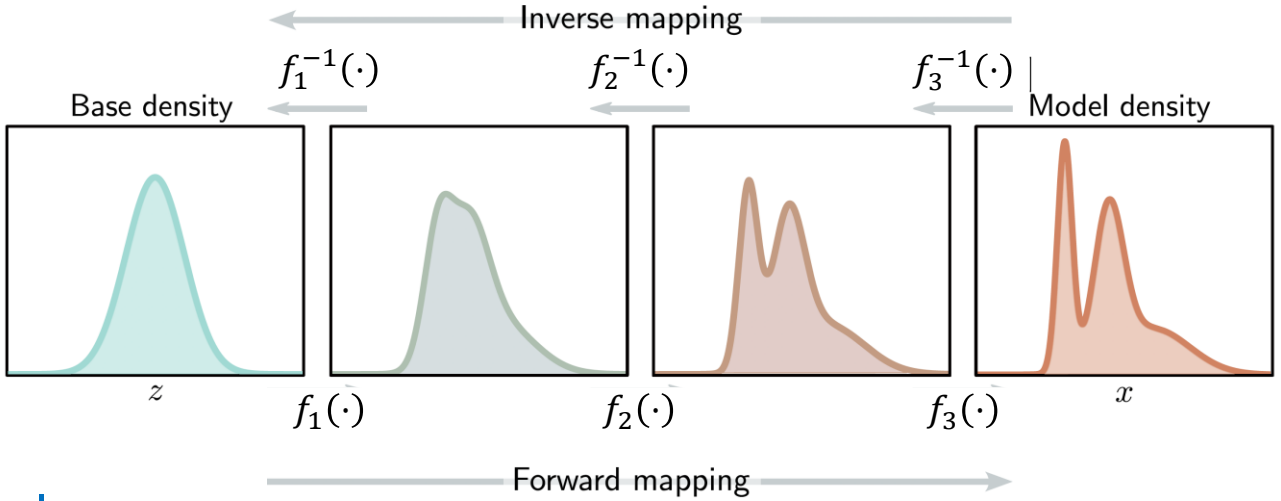
General Multistep Case

Density of the input

Used to "know" the likelihood (e.g. learning, anomaly detection)

$\boldsymbol{z_0 = f_1^{-1}(z_1)}$

$$P(\boldsymbol{x}) = P(\boldsymbol{z}_0)\prod_{i=1}^{N}\left|det\,\frac{\partial f_i^{-1}(\boldsymbol{z}_i)}{\partial \boldsymbol{z}_i}\right| = P(\boldsymbol{z}_0)\prod_{i=1}^{N}\left|\det J_{\boldsymbol{z}_i}(f_i^{-1})\right| \qquad \begin{matrix}\boldsymbol{z}_N = \boldsymbol{x}\\ \boldsymbol{z}_0 = \boldsymbol{z}\end{matrix}$$



Density of the sample

Used for sampling

$\boldsymbol{z_0 \sim P(z_0)}$

$$P(\boldsymbol{x}) = P(\boldsymbol{z}_0)\prod_{i=1}^{N}\left|det\,\frac{\partial f_i(\boldsymbol{z_{i-1}})}{\partial \boldsymbol{z_{i-1}}}\right|^{-1} = P(\boldsymbol{z}_0)\prod_{i=1}^{N}\left|\det J_{\boldsymbol{z}_{i-1}}(f_i)\right|^{-1} \qquad \begin{matrix}\boldsymbol{z}_N = \boldsymbol{x}\\ \boldsymbol{z}_0 = \boldsymbol{z}\end{matrix}$$

Some considerations & desiderata

- Can use log densities for stability and learning

$$\log P(\boldsymbol{x}) = \log P(\boldsymbol{z}_0) + \sum_{i=1}^{N} \log\left|\det J_{\boldsymbol{z}_{i-1}}\left(f_i^{-1}\right)\right| = \log P(\boldsymbol{z}_0) - \sum_{i=1}^{N} \log\left|\det J_{\boldsymbol{z}_i}(f_i)\right|$$

- Can optimize the parameters θ of the $f_i(\cdot\,;\theta)$ by gradient based optimization of the log-likelihood above
  - f_i needs to be invertible and differentiable (and remain so throughout learning)
  - f_i composition needs to be expressive enough to map Normal into arbitrary distributions
  - Need to compute determinant easily (e.g. Jacobian diagonal or triangular matrix)
  - Computation of f_i needs to be efficient for sampling
  - Computation of f_i^{-1} needs to be efficient for learning
  - Computation of f_i needs to be stable numerically

Normalizing Flow Layers

Invertible neural layers - Linear

- Affine (i.e. linear) flows can be invertible

$$\boldsymbol{f}(\boldsymbol{z}) = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{z}$$

- $\boldsymbol{W}$ needs to be full rank ⇒ expensive to compute determinant in the general case
- Interesting special cases
  - $\boldsymbol{W}$ diagonal (poorly expressive)
  - $\boldsymbol{W}$ triangular (quadratic cost to invert; still poorly expressive)
  - $\boldsymbol{W}$ permutation (volume preserving (det = 1); often used jointly with other flow layers)

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Invertible neural layers – Pointwise nonlinear

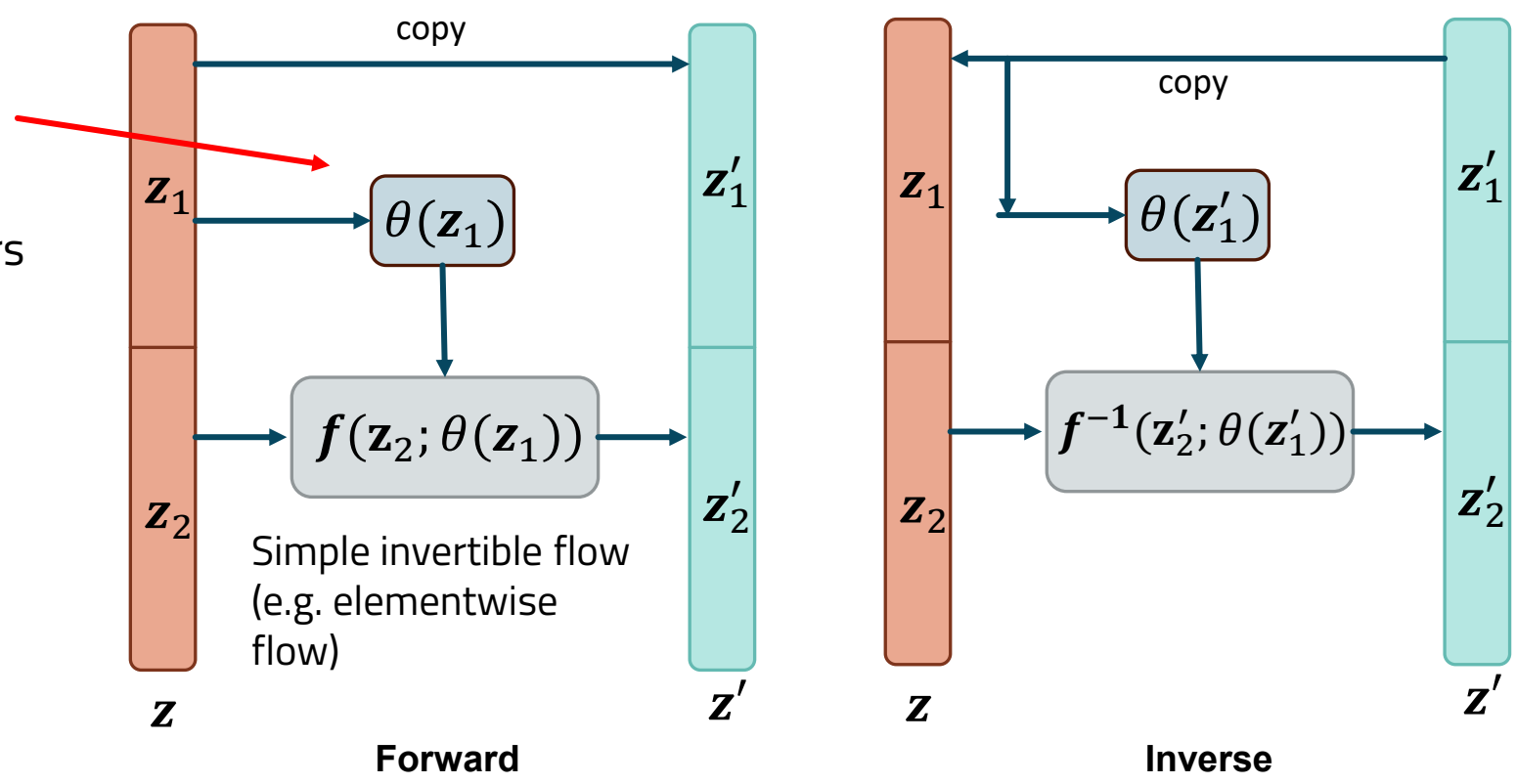
- Pointwise nonlinear where f are piecewise linear or smooth splines (nonlinear and easy to compute)

$$\boldsymbol{f}(\boldsymbol{z}) = \left[f\left(z^{(1)}, \theta\right), f\left(z^{(2)}, \theta\right), \ldots, f\left(z^{(D)}, \theta\right)\right]$$

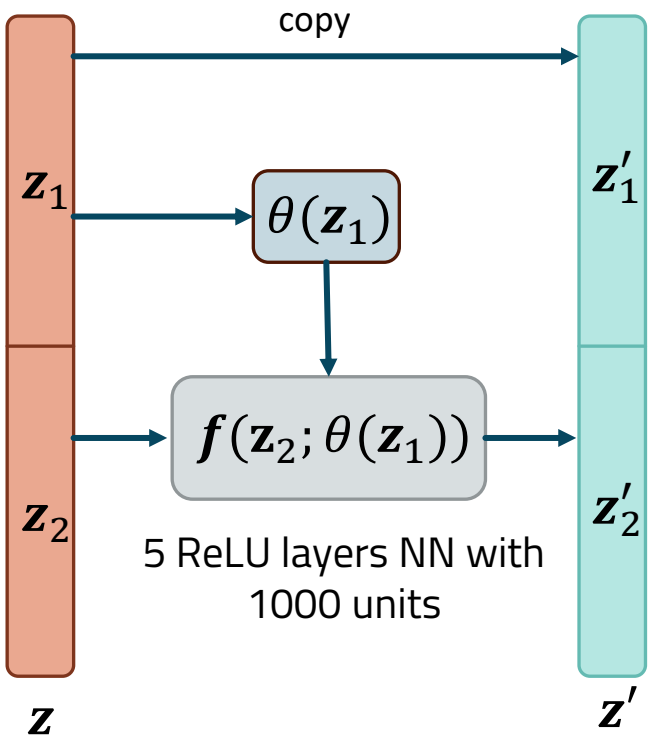
- If f is invertible, so it is $\boldsymbol{f}$
- Diagonal Jacobian
- Pointwise does not allow capturing correlations between dimensions

Coupling Flows

Neural network that generates parameters θ for the invertible function



Non-linear Independent Components Estimation (NICE)



Copy Shift

$$z'_2 = z_2 + \theta(z_1) \longrightarrow z_2 = z'_2 - \theta(z'_1)$$

Inverse

Jacobian

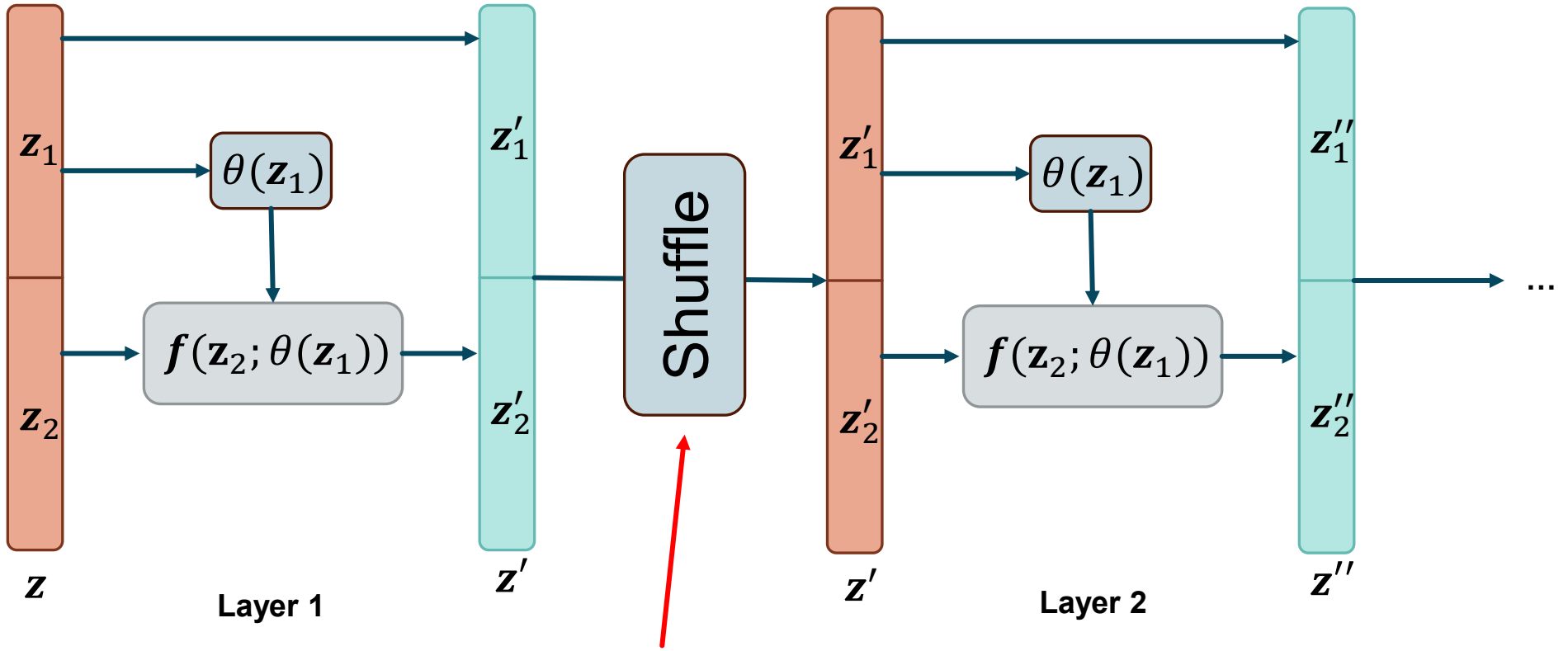
$$\begin{pmatrix} I & 0 \\ \frac{\partial \theta(z_1)}{\partial z_1} & I \end{pmatrix}$$

First example of coupling layer

- Simple affine transformation

L Dinh et al, Non-linear Independent Components Estimation (NICE), ICLR-WS 2014

NICE – Stacked Coupling Flows



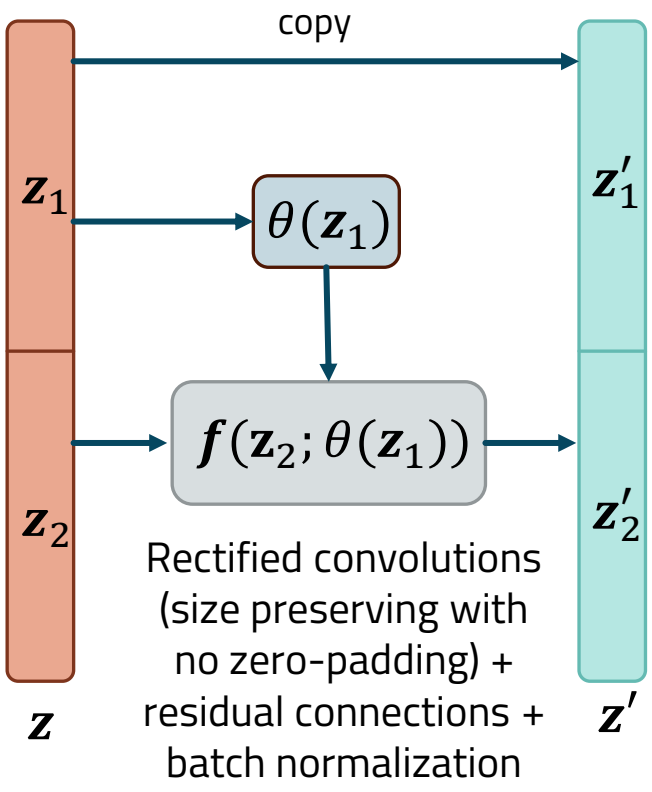
Random shuffle allows more general transformations than between only elements in 1st and 2nd half

(Not so) NICE Results

(a) Model trained on MNIST (b) Model trained on TFD (c) Model trained on SVHN (d) Model trained on CIFAR-10

L Dinh et al, Non-linear Independent Components Estimation (NICE), ICLR-WS 2014

RealNVP – Multiscale Nonlinear Flow



Scale

Shift

$$\mathbf{z}_2' = \exp(\boldsymbol{\theta}_A(\boldsymbol{z}_1)) \odot \boldsymbol{z}_2 + \boldsymbol{\theta}_B(\boldsymbol{z}_1) \quad \textbf{(Forward)}$$

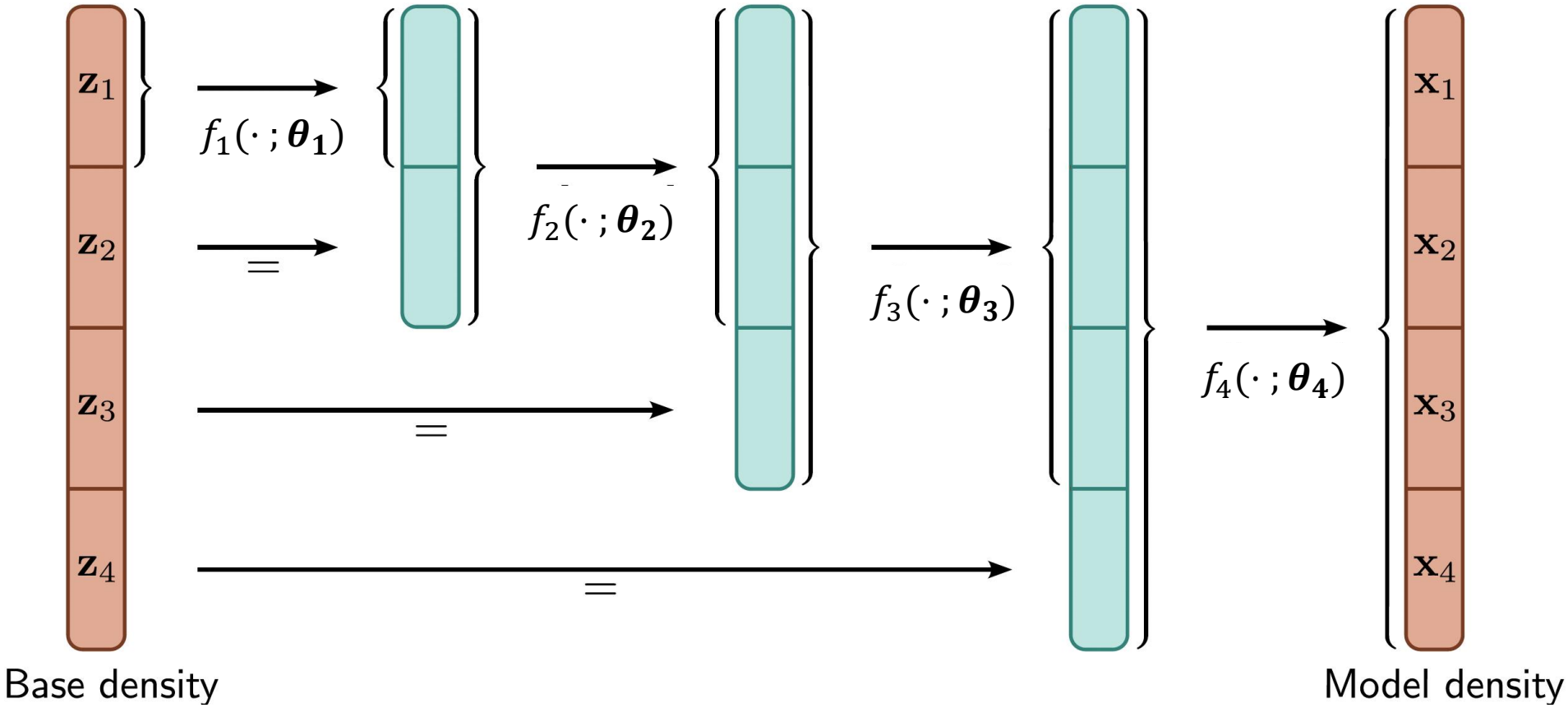
Jacobian

$$\begin{pmatrix} \boldsymbol{I} & \mathbf{0} \\ \dfrac{\partial \boldsymbol{\theta}_b(\boldsymbol{z}_1)}{\partial \boldsymbol{z}_1} & diag\ \exp(\boldsymbol{\theta}_A(\boldsymbol{z}_1)) \end{pmatrix}$$

$$\mathbf{z}_2 = \frac{\mathbf{z}_2' - \boldsymbol{\theta}_B(\boldsymbol{z}_1')}{\exp(\boldsymbol{\theta}_A(\boldsymbol{z}_1'))} \quad \textbf{(Inverse)}$$

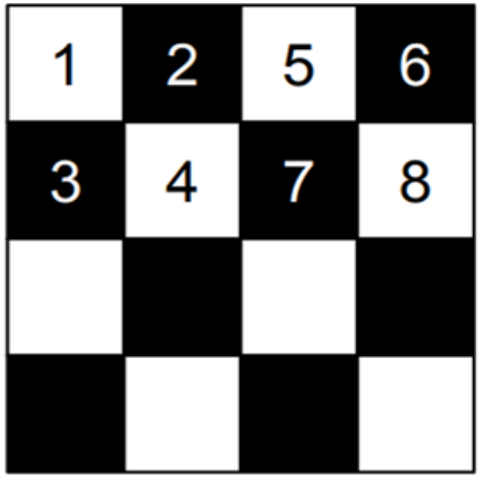
L Dinh et al, Density Estimation using real NVP, ICLR 2017

Multiscale Flows

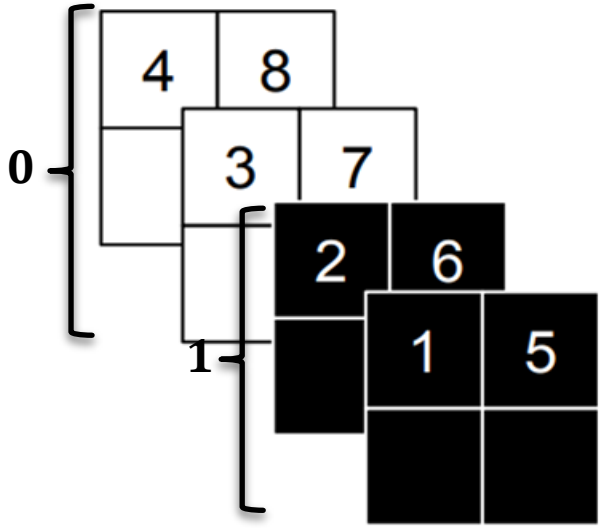


RealNVP – Masking and Squeezing

$$\boldsymbol{z}' = \boldsymbol{b} \odot \boldsymbol{z} + (1 - \boldsymbol{b}) \odot \{\exp(\boldsymbol{\theta}_A(\boldsymbol{b} \odot \boldsymbol{z})) \odot \boldsymbol{z} + \boldsymbol{\theta}_B(\boldsymbol{b} \odot \boldsymbol{z})\}$$



Partitioning
with checkerboard pattern



Squeezing
followed by channel-wise masking

$$s \times s \times c \xrightarrow[\text{Squeezing}]{} \frac{s}{2} \times \frac{s}{2} \times 4c$$

Multiscale flow implemented by alternating binary masking ($b_i \in \{0,1\}$) and squeezing

- Pixel masking before squeeze
- Channel masking after squeeze

L Dinh et al, Density Estimation using real NVP, ICLR 2017

RealNVP Results

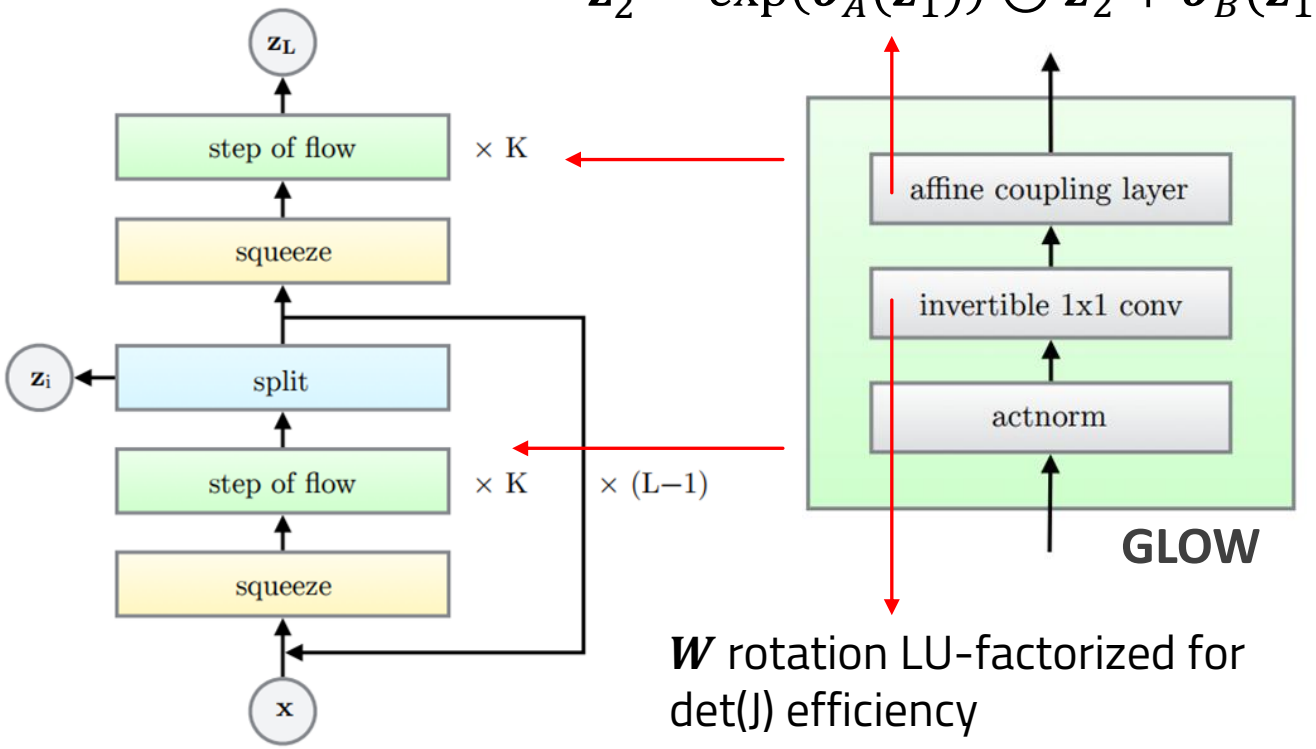
L Dinh et al, Density Estimation using real NVP, ICLR 2017

dataset

sampled

GLOW – Multiscale Coupling Flow with Invertible 1x1 Convolutions

$$\mathbf{z}_2' = \exp(\boldsymbol{\theta}_A(\boldsymbol{z}_1)) \odot \boldsymbol{z}_2 + \boldsymbol{\theta}_B(\boldsymbol{z}_1)$$

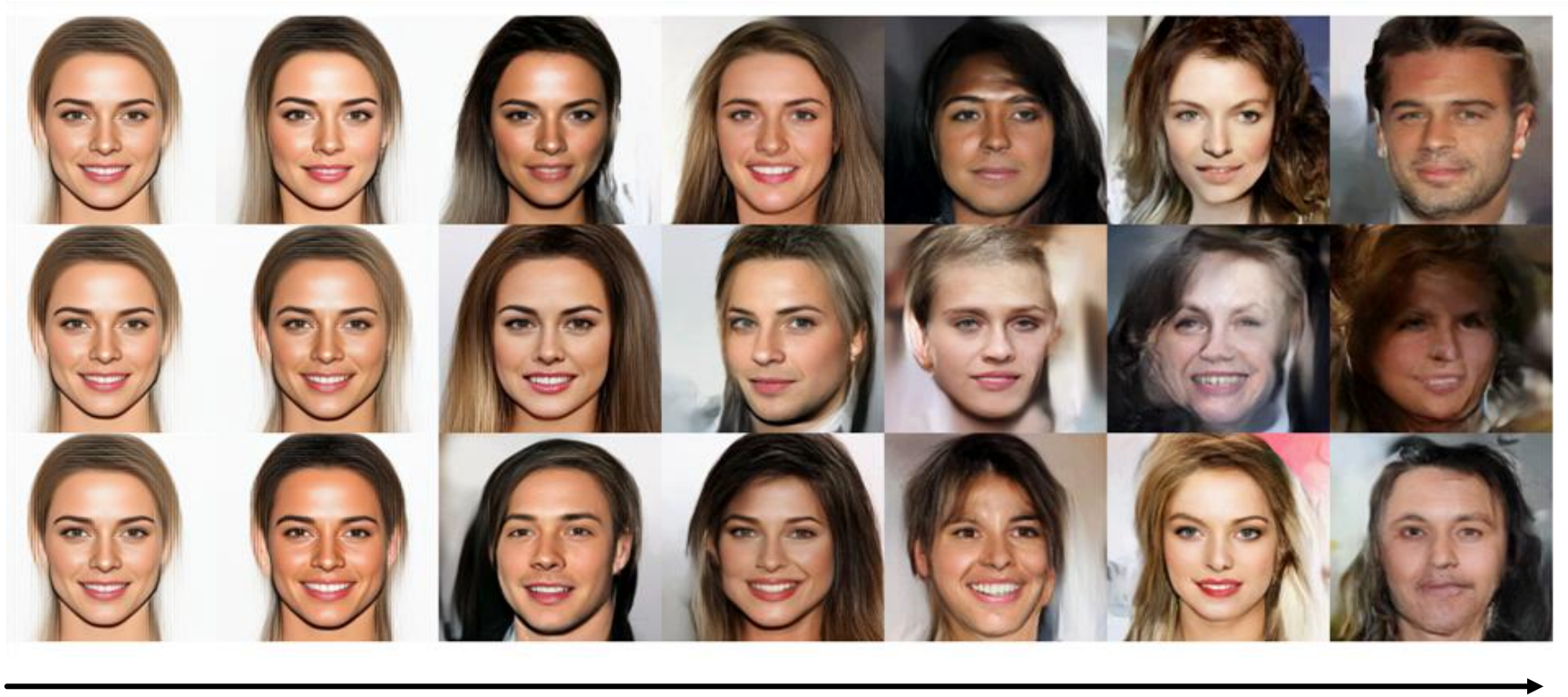


$\boldsymbol{W}$ rotation LU-factorized for det(J) efficiency

- Start with RGB tensor
- Split channels in 2 halves
- Run 1x1 convolutions parameterized with an LU decomposition (channel mixing/permutation)
- Affine transform each spatial position in second half
- Multiscale & periodic squeeze

Kingma & Dhariwal, P, Glow: Generative flow with invertible 1x1 convolutions, NeurIPS 2018

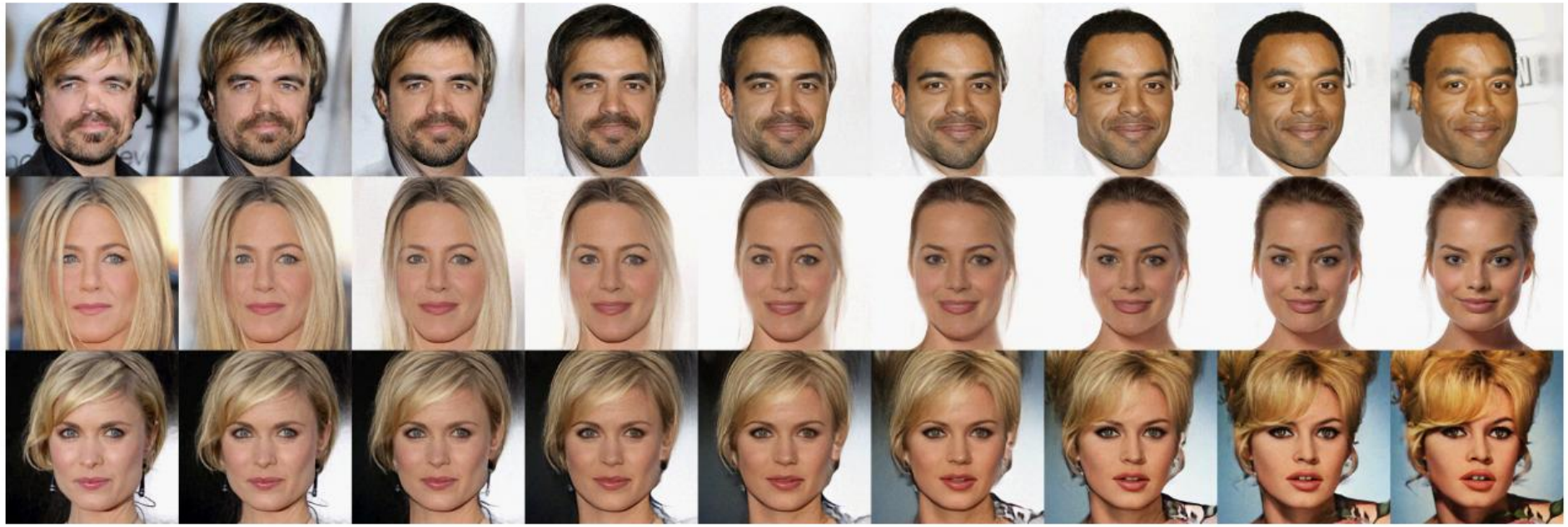
GLOW Results - Sampling



Increasing temperature

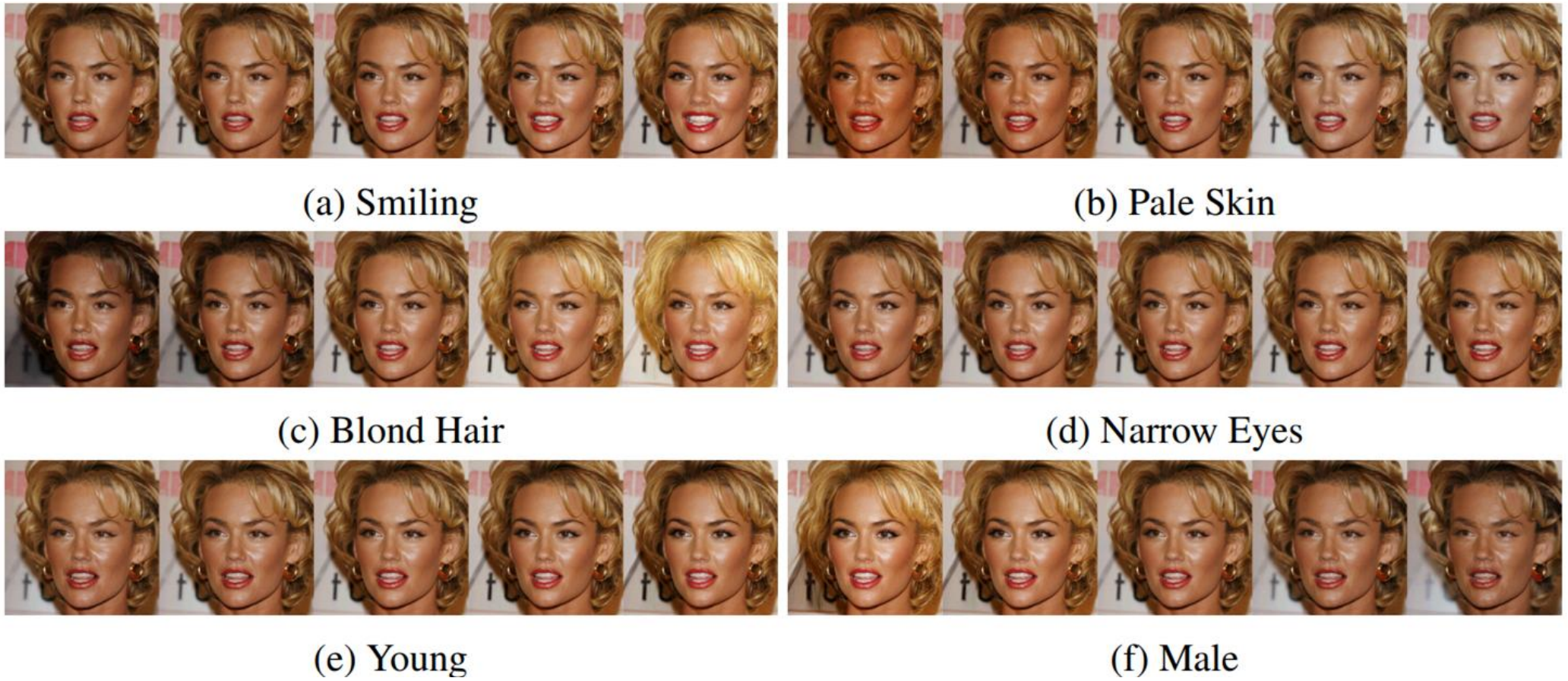
Kingma & Dhariwal, P, Glow: Generative flow with invertible 1x1 convolutions, NeurIPS 2018

GLOW Results - Interpolation



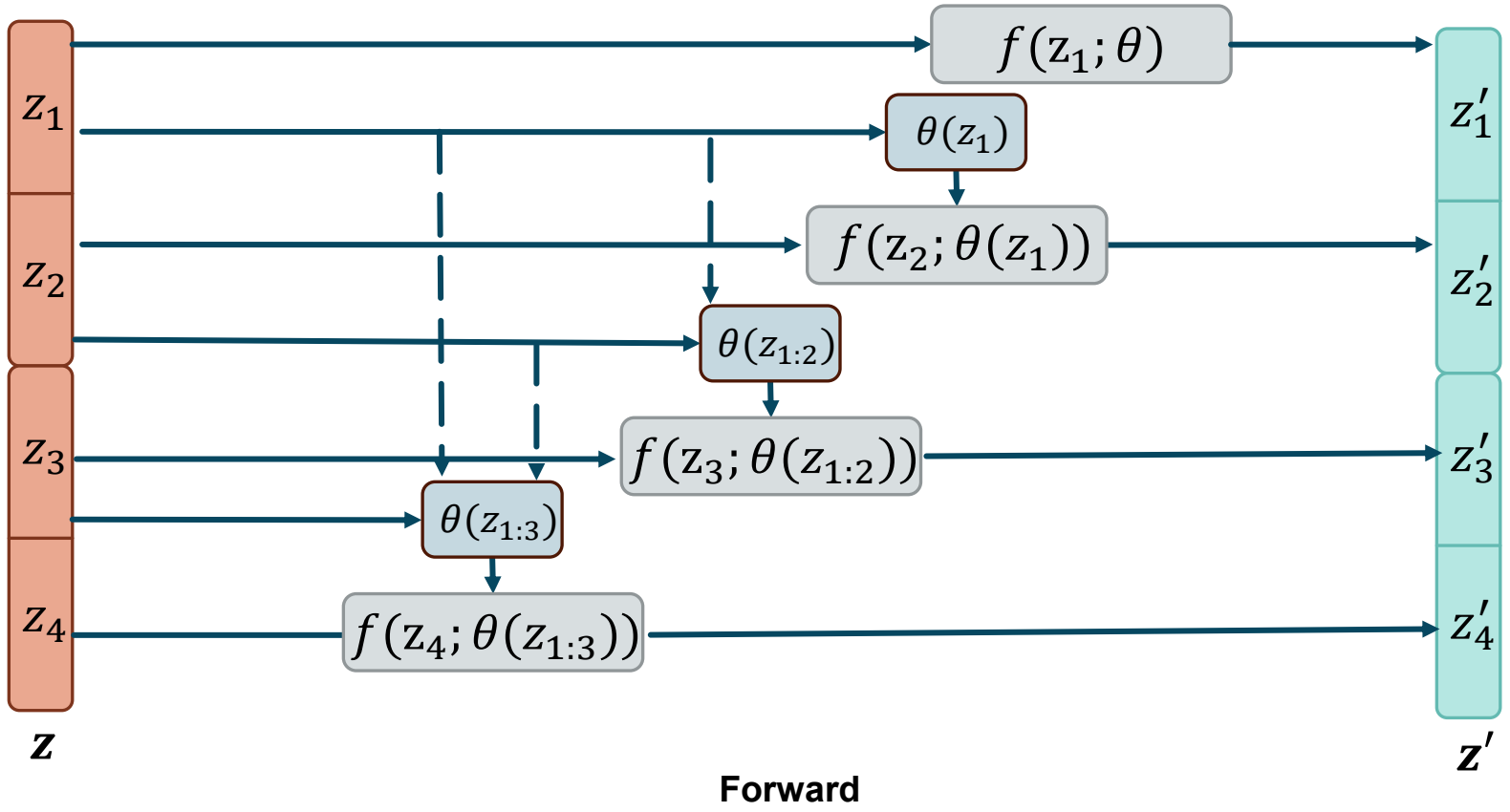
Kingma & Dhariwal, P, Glow: Generative flow with invertible 1x1 convolutions, NeurIPS 2018

GLOW Results - Manipulation



Kingma & Dhariwal, P, Glow: Generative flow with invertible 1x1 convolutions, NeurIPS 2018

Autoregressive Flows

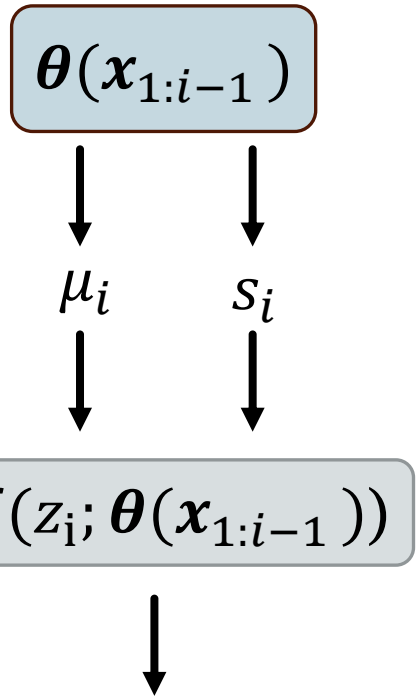


Generalization of coupling flows that treats each input dimension as a separate block

Forward and inverse directions have different costs (parallel/sequential)

Masked Autoregressive Flow

G. Papamakarios et al, Masked Autoregressive Flow for Density Estimation, NeurIPS 2017



$$x_i = \mu_i + z_i \exp(s_i)$$
$$z_i \sim \mathcal{N}(0,1)$$

Autoregressive model as a transformation $\boldsymbol{f}$ from the space of random vectors $\boldsymbol{z}$ to space of data $\boldsymbol{x}$

$$P(x_i | \boldsymbol{x}_{1:i-1}) = \mathcal{N}(\mu_i, (\exp(s_i))^2)$$

$\boldsymbol{f}$ is easily invertible, Jacobian is triangular and easily computable determinant

$$z_i = f^{-1}(x_i) = (x_i - \mu_i) \exp(-s_i)$$
$$\left| \det \left(\frac{\partial f^{-1}}{\partial z} \right) \right| = \exp - \sum_i s_i$$

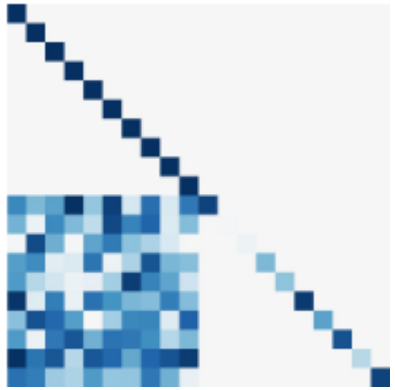
Residual flows

Designing scalable flows

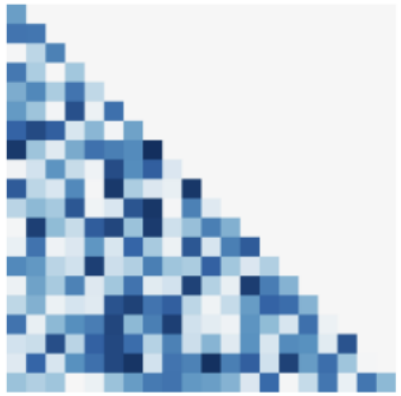
So far, to keep Normalizing Flows scalable we have restricted our layers to have "well" behaved Jacobians

$$\log P(\boldsymbol{x}) = \log P(f(\boldsymbol{x})) + \log|\det J_{\boldsymbol{x}}(f)|$$

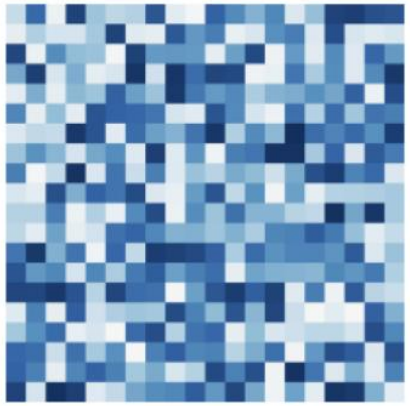
Coupling layers
NICE
Real NVP
Glow



Autoregressive
Masked
Inverse
…



Can we get a more expressive (full rank) Jacobian and still remain scalable?



Residual flows

Use a residual-based neural layer

$$\boldsymbol{z}' = \boldsymbol{z} + \theta(\boldsymbol{z})$$

Intuition

- Small updates allow for easier invertibility
- Composing many small updates (i.e. using a deep network) yields expressive transformations

Formally

- If θ is Lipschitz with constant $L < 1$

$$||\theta(\boldsymbol{z}) - \theta(\boldsymbol{z}')|| \leq L\,||\boldsymbol{z} - \boldsymbol{z}'||$$

- Then by Banach fixed-point $\boldsymbol{z}' = \boldsymbol{z} + \theta(\boldsymbol{z})$ is invertible

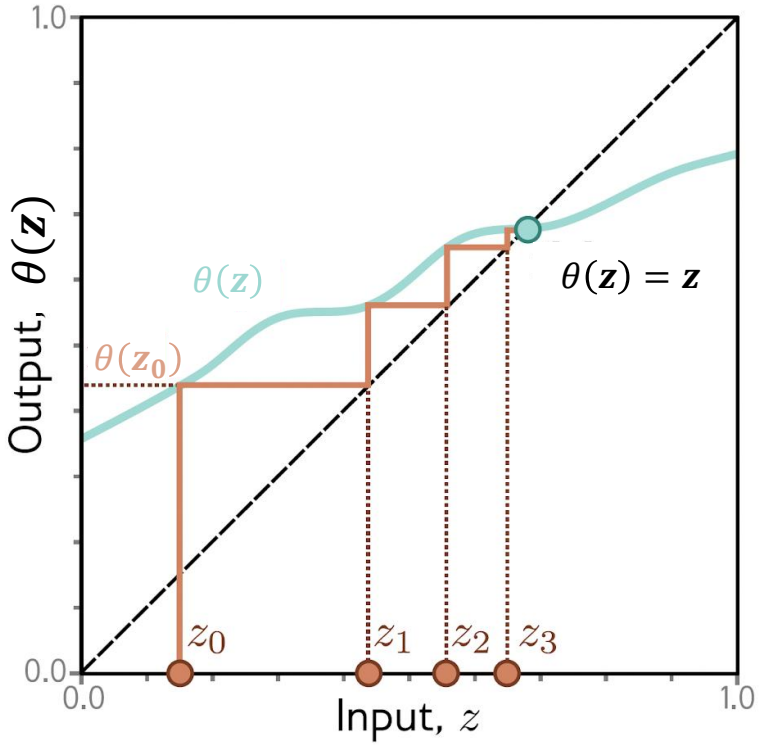
In practice

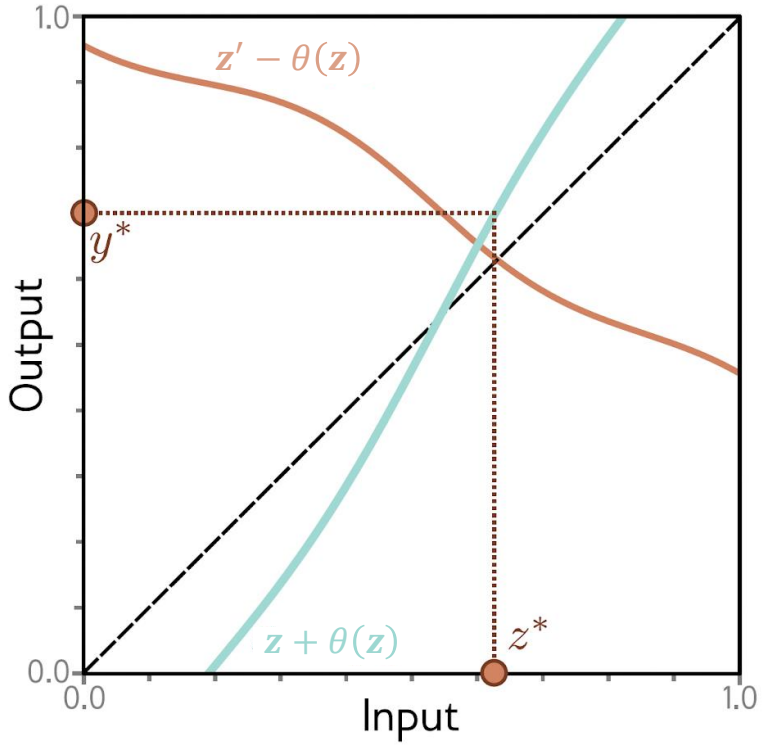
The inverse can be obtained via fixed-point iterative method

$$\boldsymbol{z}^{(i)} = \boldsymbol{z}' - \theta(\boldsymbol{z}^{(i-1)})$$

until it converges to the $\boldsymbol{z}^*$ that is the starting point for $\boldsymbol{z}'$

Contration Mappings

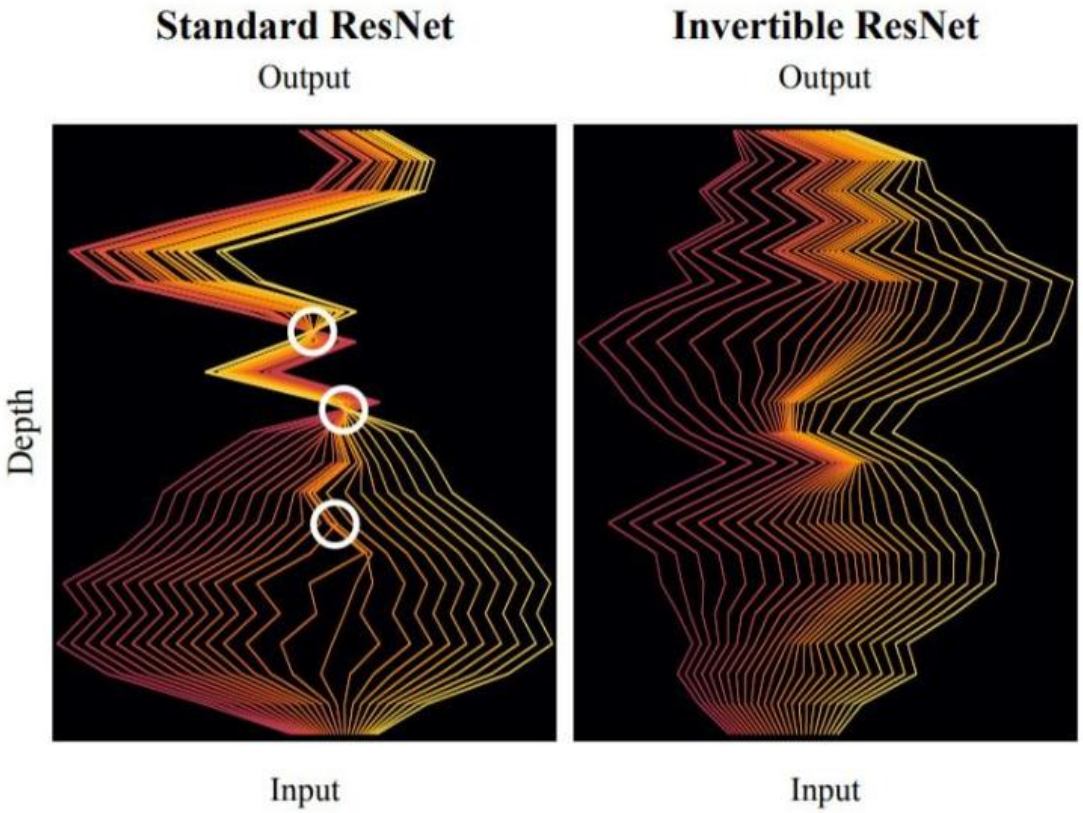




When a contractive mapping is iterated (i.e., the output is repeatedly passed back in as an input), the result converges to a fixed point where $\theta(\boldsymbol{z}) = \boldsymbol{z}$

This can be done by starting with any point $\boldsymbol{z_0}$ and iterating $\boldsymbol{z}^{(i)} = \boldsymbol{z}' - \theta\left(\boldsymbol{z}^{(i-1)}\right)$, which has a fixed point at $\boldsymbol{z} + \theta(\boldsymbol{z}) = \boldsymbol{z}'$

Invertible ResNets



Obtaining invertible residual layers entails

- Choosing Lipschitz-constrained activation functions (i.e. $\sigma'(\cdot) < 1$)
- Bounding the spectral radius of weight matrices

Invertible ResNets and their Jacobian

- For an invertible residual change of variable

$$\boldsymbol{z}' = f(\boldsymbol{z}) = \boldsymbol{z} + \theta(\boldsymbol{z})$$

- The target log-likelihood rewrites as

$$\log P(\boldsymbol{z}) = \log P(\boldsymbol{z}') + \log|\det J_{\boldsymbol{z}}(f)| = \log P(\boldsymbol{z}') + \left(\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k}\, tr(\boldsymbol{J}_{\boldsymbol{z}}(\boldsymbol{\theta}))^k\right)$$

Can be approximated efficiently (linearly) using the Hutchinson trace estimator $tr(\boldsymbol{A}) = \mathbb{E}[\boldsymbol{\epsilon}^T \boldsymbol{A}\boldsymbol{\epsilon}]$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$

One final twist

Let's reformulate the invertible residual adding a scaling $\delta > 0$

$$\boldsymbol{z}' = f(\boldsymbol{z}) = \boldsymbol{z} + \delta\theta(\boldsymbol{z})$$

Rearranging terms

$$\theta(\boldsymbol{z}) = \frac{\boldsymbol{z}' - \boldsymbol{z}}{\delta}$$

which looks a lot like a derivative.

We can make the thing more explicit by defining the infinitesimal update (for $\Delta t = \delta \to 0$)

$$\boldsymbol{z}_{t+\delta} = \boldsymbol{z}_t + \delta \frac{\partial \boldsymbol{z}_t}{\partial t} \text{ s.t. } \underbrace{\frac{\partial \boldsymbol{z}_t}{\partial t} = \theta_t(\boldsymbol{z}_t)}_{\text{Instantaneous change}}$$

Continuous Normalizing Flows

Now suppose we are cascading multiple residual layers $f_i(\cdot)$, yielding the composition

$$f_T \circ f_{T-1} \circ \cdots \circ f_1$$

Using the instantaneous update defined in previous slide this can be represented by an Ordinary Differential Equation (ODE)

A neural ODE →

$$\begin{cases} \boldsymbol{z}_0 \sim N(0,1) \\ \dfrac{\partial \boldsymbol{z}_t}{\partial t} = \theta_t(\boldsymbol{z}_t) \end{cases}$$

Initial conditions

Instantaneous update (vector field)

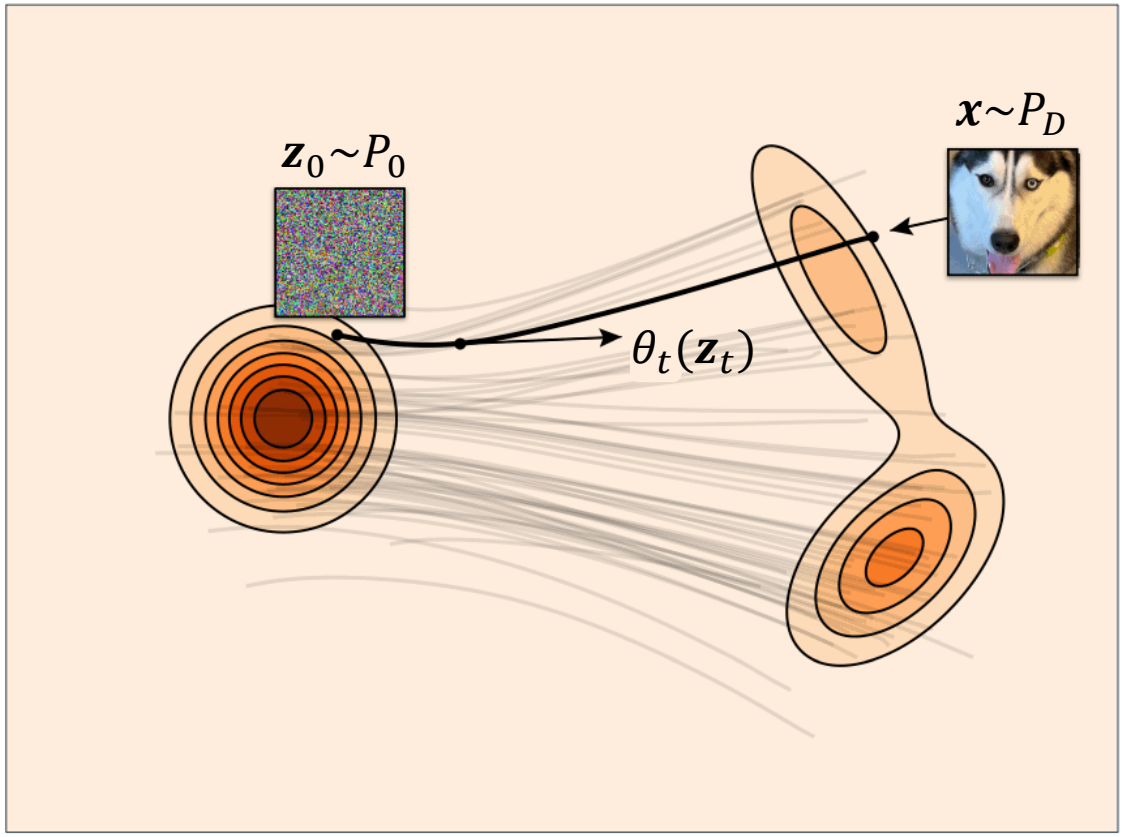
Plugging the continuous formulation above in the change of variable setting of Normalizing Flows gives us the loglikelihood for instantaneous change of variables

$$\log P(\boldsymbol{x}) = \log P(\boldsymbol{z}_0) - \int_0^T Tr\left(\boldsymbol{J}_{\boldsymbol{z}_t}(\theta_t)\right) dt$$

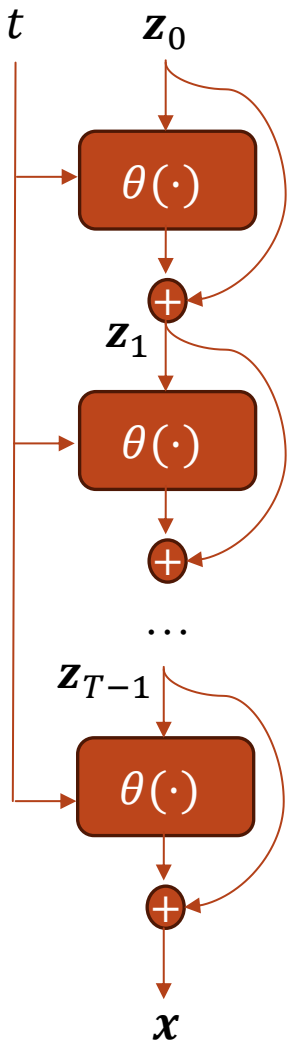
No Jacobian determinant! Change in log probability is related to the trace of the derivative of the forward propagation

Which can be solved using numerical integration

Morphing distributions by continuous change of variable



Continuous change of variable trajectories



Discretized ODE as a ResNet

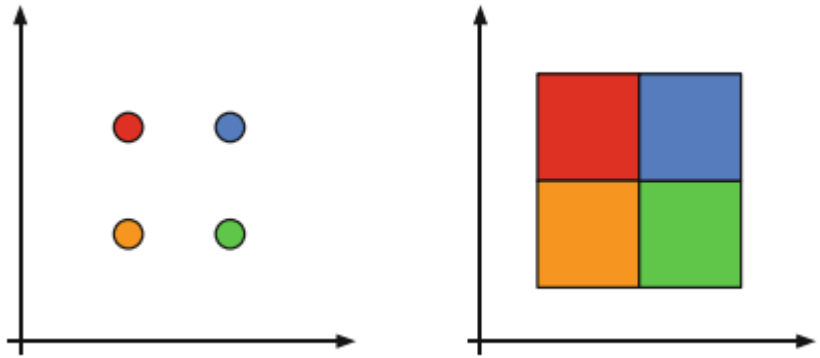
Wrap-up

Implementations & Libraries

- Normalizing flows are natively supported by Tensorflow (through the TF Probability module)
  - tf.probability.distribution (for base distributions)
  - tf.probability.bijector (for predefined layers, e.g. masked autoregressive)
  - Chain() (to chain bijectors and compose complex modules)
  - You can of course define you own bijectors according to a template
- Normflows - PyTorch package for Normalizing Flows
- Flowtorch – PyRo based Pytorch library for Normalizing Flows

A pragmatic note - Dequantization

- Flow-based models assume that **x** is a vector of real-valued random variables
- In practice, **x** is often many objects discrete valued (e.g. pixel colors have values in {0,…,255})
- Uniform dequantization - Add uniform noise $\boldsymbol{u} \in [-0.5, 0.5]^d$ to original data $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{u}$
- Dequantization is supported by TF/Torch APIs



Take Home Messages

- Normalizing flows as an effective and tractable way to generate new samples (efficient) and to evaluate the likelihood of samples (not so efficient)
- Universality property - The flow can learn any target density to any required precision given sufficient capacity and data
  - Flow can be used to generate samples that approximate a density easy to evaluate but difficult to sample
- Normalizing flow design needs to take care of
  - Keeping flow invertible and efficient
  - Making the determinant of the Jacobian easy to compute
- Normalizing flows can be made continuous using a neural ODE scheme
  - Can be implemented using ResNets
  - Expressive and powerful: can parameterize a large class of probability distributions
  - Can be extremely slow due to ODE numerical integration

We may be doing something about this in a couple of lectures

Next Lecture

No lecture on 05/05/2026

Diffusion models

- Generative learning as a noising-denoising process
- Guided diffusion
- Conditional diffusion
- Latent space diffusion