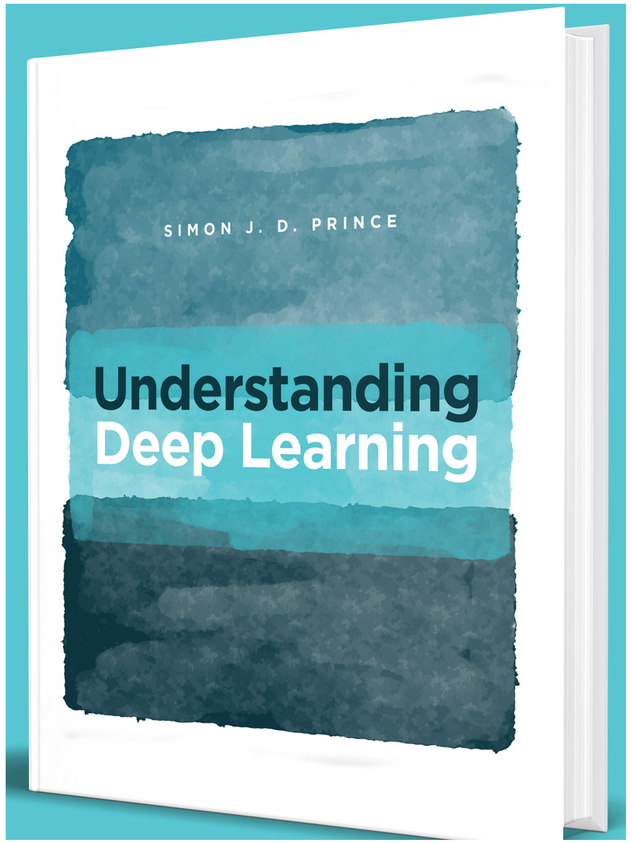
Lecture 12: Normalizing Flows

Maurizio Pierini (CERN), Thea Aarrestad (ETH)

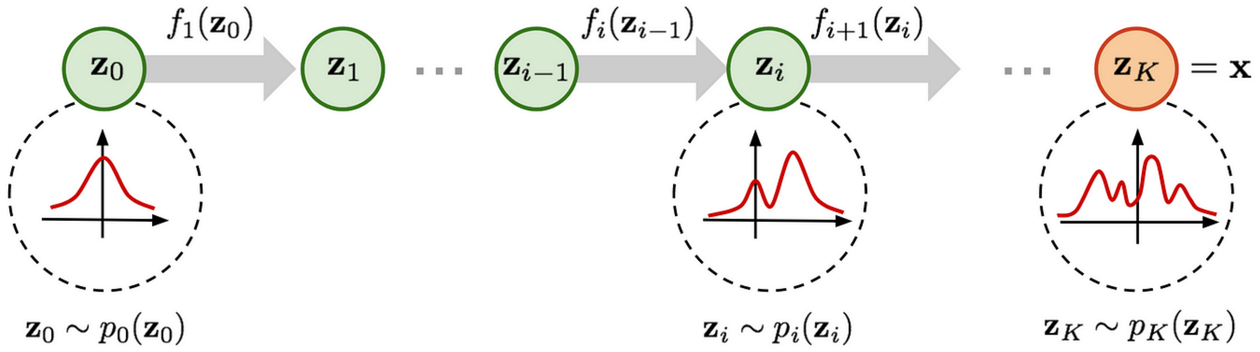
Program for Today



A	B	C
Date	Topic	Tutorial
Sep 16	Intro & class description	
Sep 23	probability Theory + Basic of machine learning + Dense NN	*Basic jupyter + DNN on mnist (give jet dnn as homework)*
Sep 30	Statistics + Convolutional NN	*Convolutional NNs with MNIST*
Oct 7	Training in practice: regularization, optimization, etc	*Free Practice*
Oct 14	Unsupervised learning and anomaly detection	*Autoencoders with MNIST*
Oct 21	Graph NNs	*Tutorial on Graph NNs*
Oct 28	Transformers	
Nov 4	Network compression (pruning, quantization, Knoweldge Distillation)	
Nov 11	Tutorial on hls4ml	
Nov 18	Generative models: GANs, VAEs, etc	*GAN and VAE*
Nov 25	Reinforcement Learning	
Dec 2	Normalizing flows	*Normalizing flows*
Dec 9	Quantum Machine Learning	
Dec 16	**Exams To Be Confirmed**	

Going Beyond GANs (and VAEs)

- GANs are trained to create new realistic examples learning from a given dataset
- But they gain no knowledge on the distribution over the data sample
  - Because of that, they might generate images of one kind more than another, w/o respecting the relative population of the reference dataset
- We saw that VAEs are trained to “force” this knowledge, imposing a certain prior in the latent space through the KL regularization. But they have other issues (e.g., blurry images)
- Normalizing Flows address the sampling problem:
  - They learn to transform an easy-to-control distribution (e.g., Gaussian) into an arbitrary one (i.e., they fit the reference data distribution with a neural network)
    - GENERATORS: One can then sample a generic function, sampling the easy one and applying the morphing network
    - ANOMALY DETECTION: One can associate a likelihood p-value to a given event



Invertible Change of Variable

- We change variable from z to x, transporting an infinitesimal of probability density using a bijector (invertible and differentiable function)

$x = f(z)$

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

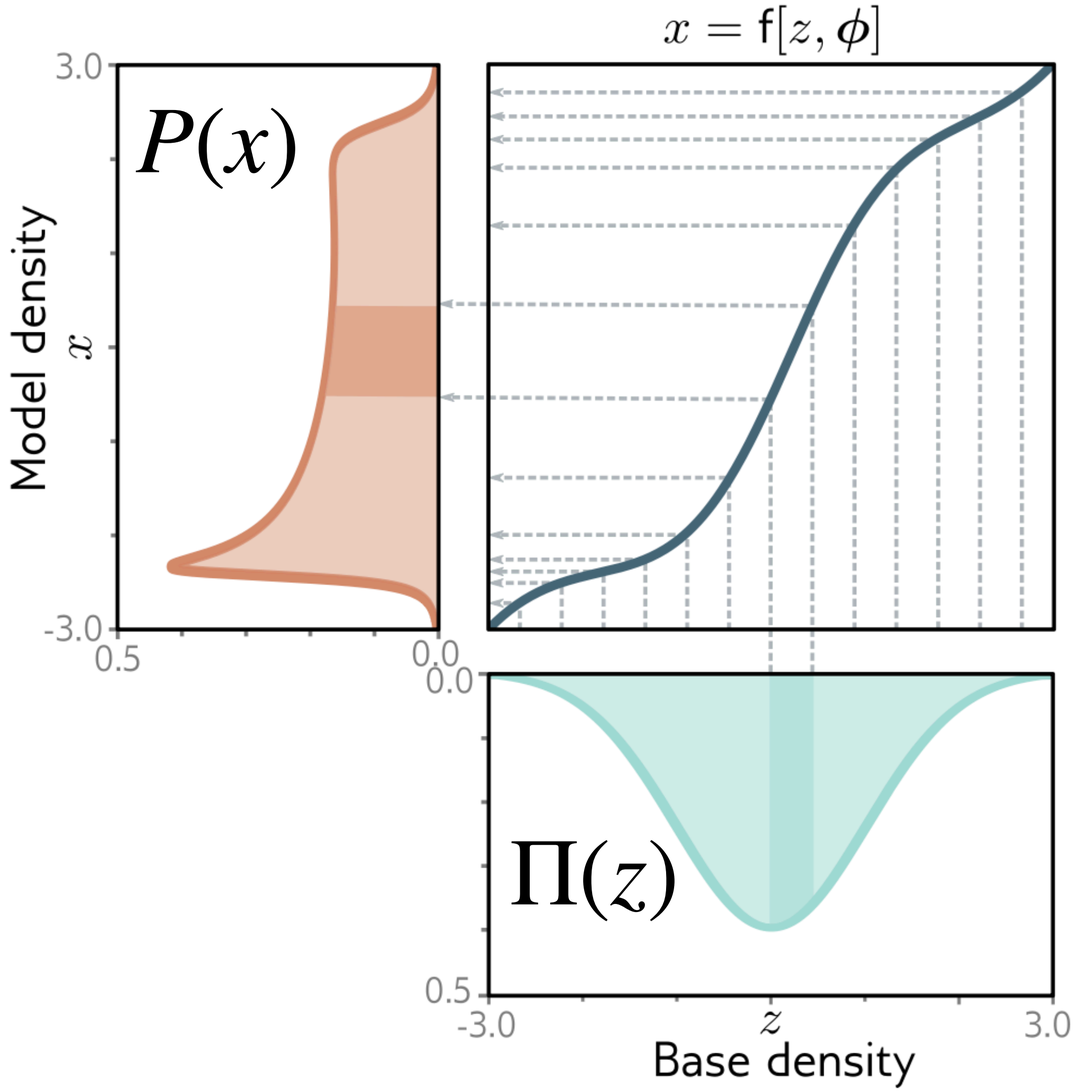
- But this implies

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

where the last equality follows from the inverse function theorem

- For normalizing flows $x = f(z|\phi)$, where ϕ are the parameters of the neural network f.

This implies that $P(x|\phi) = \Pi(z)\left|\frac{\partial f(z|\phi)}{\partial z}\right|^{-1}$



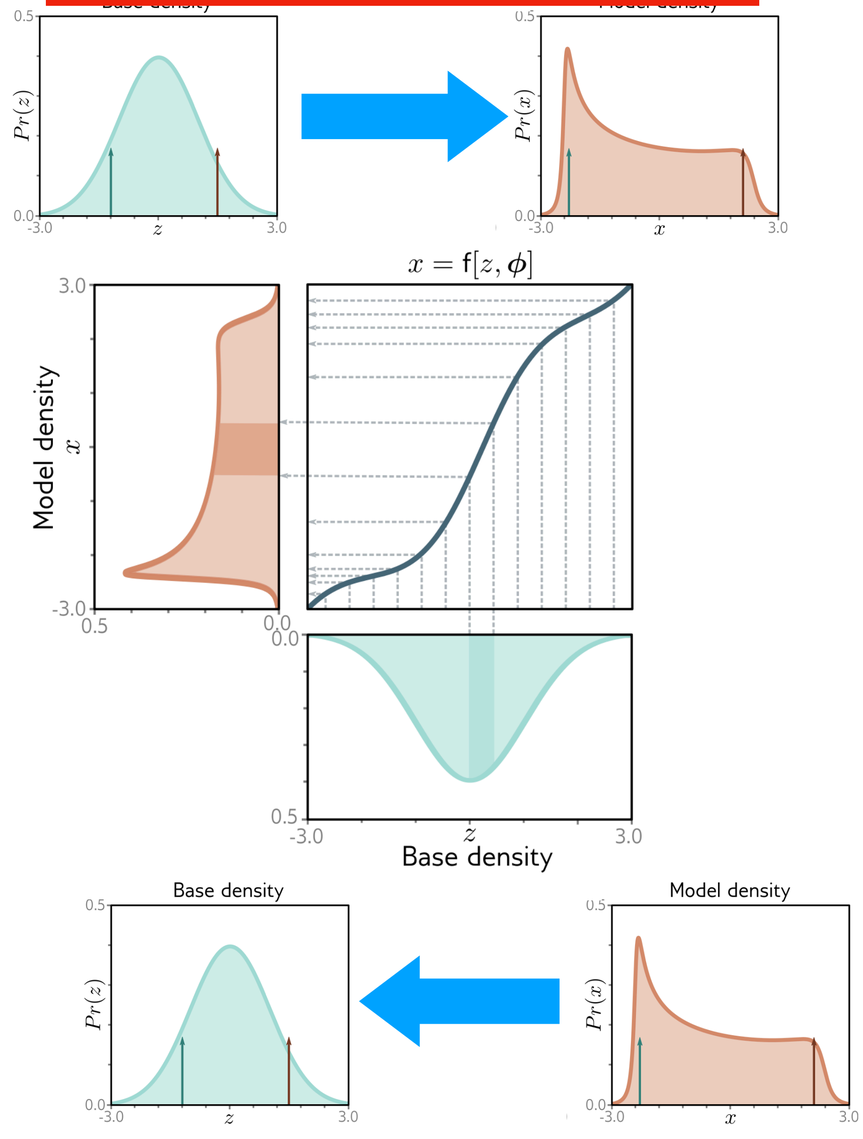
1D Normalizing Flow

- Start with a Gaussian $G(z)$
- Apply a transformation $x = f(z)$ so that x is distributed according to the desired function

$$P(x \mid \phi) = \left| \frac{\partial f(z, \phi)}{\partial z} \right|^{-1} \Pi(z)$$

  - Notice that depending on the Jacobian being > or < 1, probability is compressed or stretched
  - The transformation is reversible → one can do the opposite and map the reference distribution back to the latent one
    - This is useful for anomaly detection: can associate a probability (technically, a p-value) to a point in x without knowing P(x)

CAN DRAW SAMPLES FROM DISTRIBUTION



CAN MEASURE LIKELIHOOD

Training a Normalizing Flow

- A Normalizing Flow is trained looking for the parameters of the network that maximize the likelihood of the training data distribution P(x)
  - It is just a likelihood fit, but done through NN

$$\hat{\phi} = \text{argmax}_{\phi}\left[\prod_i P(x_i|\phi)\right] = \text{argmin}_{\phi}\left[-\log\prod_i P(x_i|\phi)\right] = \text{argmin}_{\phi}\left[\log\prod_i \left|\frac{\partial f(z_i,\phi)}{\partial z_i}\right| - \log\prod_i \Pi(z_i)\right] =$$

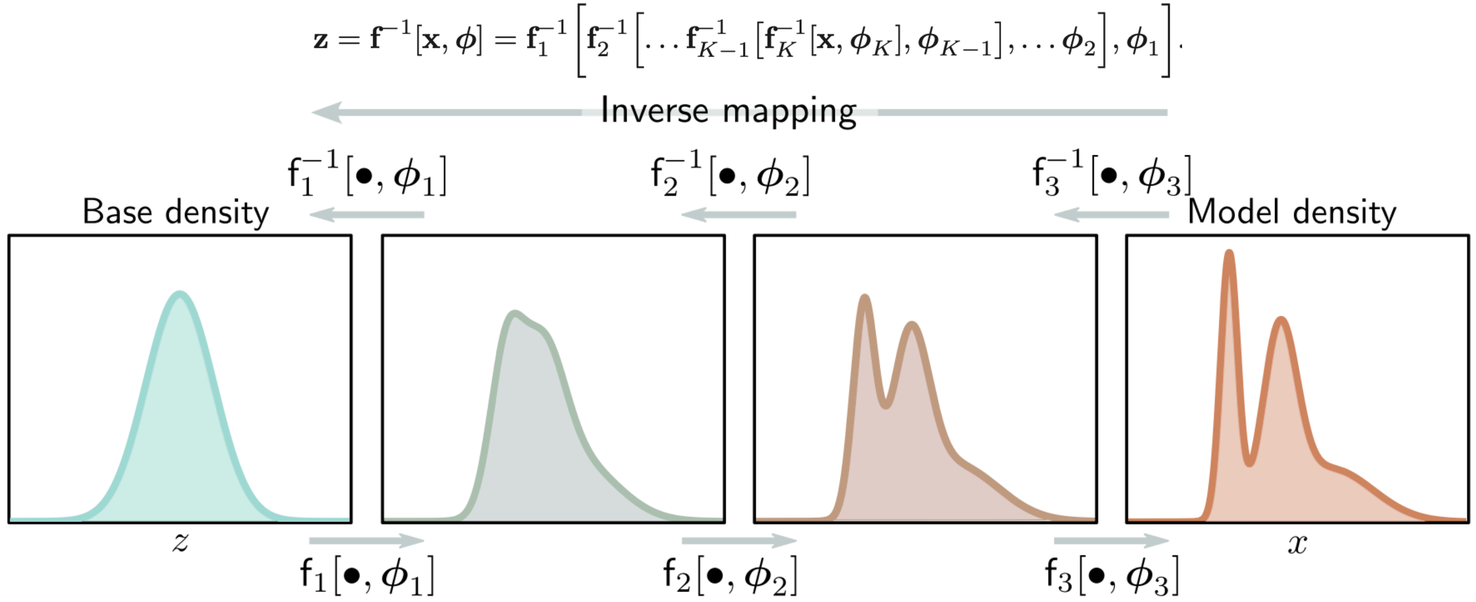
$$\text{argmin}_{\phi}\left[\sum_i \log\left|\frac{\partial f(z_i,\phi)}{\partial z_i}\right| - \sum_i \log\Pi(z_i)\right]$$

- One can easily generalize this to the case of N-dim, replacing the abs value of the partial derivative with the determinant of the Jakobian

A Deep Normalizing Flow

- *A deep NF is a chain of NFs, gradually morphing the latent function $\Pi(z)$ into the target function $P(x)$*

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



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

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

TRAINING

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\prod_{i=1}^{I} Pr(\mathbf{z}_i) \cdot \left|\frac{\partial \mathbf{f}[\mathbf{z}_i, \boldsymbol{\phi}]}{\partial \mathbf{z}_i}\right|^{-1}\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmin}} \left[\sum_{i=1}^{I} \log\left[\left|\frac{\partial \mathbf{f}[\mathbf{z}_i, \boldsymbol{\phi}]}{\partial \mathbf{z}_i}\right|\right] - \log\big[Pr(\mathbf{z}_i)\big]\right]\end{aligned}$$

The NF Architecture

- *So far we did not specify what f looks like*
- *The choice of f, i.e., of the network architecture, should be made so that certain properties of f are obtained*
  - *The set of f functions chained in the deep NF should be expressive enough to be capable of learning many distributions*
  - *Each layer should be invertible (1-to-1 mappings)*
  - *The inverse should be computable efficiently*
  - *The determinant of the Jacobian should be computed efficiently*

Linear Flows

- As the name suggests, a linear flow is a transformation $f[z] = \Omega z + \beta$, which is invertible if Ω is invertible
- Linear flows are expensive at large dimensions D
  - Computing the determinant and the inverse are $\mathcal{O}[D^3]$ slow, less if matrices are special (triangular, diagonal, etc.)
- Linear flows are not very expressive: they can't learn non-linear transformations and a Gaussian stays a Gaussian

Elementwise flows

- *Elementwise flows are the simplest examples of non-linear flows*
- *They correspond to apply the same non linear functions to each element of the input*

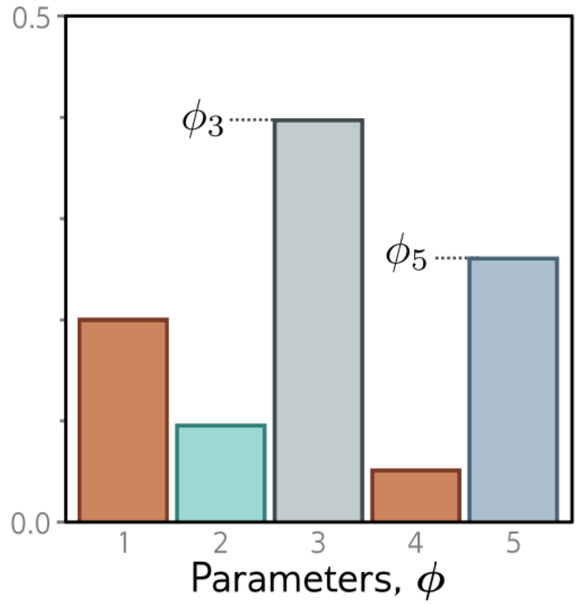
$$\mathbf{f}[\mathbf{h}] = \Big[\mathrm{f}[h_1, \boldsymbol{\phi}], \mathrm{f}[h_2, \boldsymbol{\phi}], \ldots \mathrm{f}[h_D, \boldsymbol{\phi}]\Big]^T$$

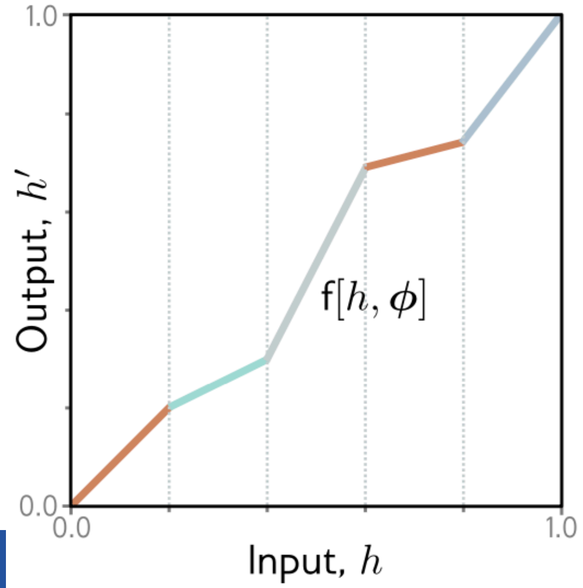
- *The Jakobian is diagonal, since there in no cross-talk between the elements in the transformation. So the determinant calculation is easy*

$$\left|\frac{\partial \mathbf{f}[\mathbf{h}]}{\partial \mathbf{h}}\right| = \prod_{d=1}^{D} \left|\frac{\partial \mathrm{f}[h_d]}{\partial h_d}\right|$$

- *It adds non-linearity, but it cannot alter correlations between input quantities*
- *The function could be any invertible non-linear function, as simple as LeakyReLu (no trainable parameters) or a generalization of that, with trainable slope (piecewise linear function)*

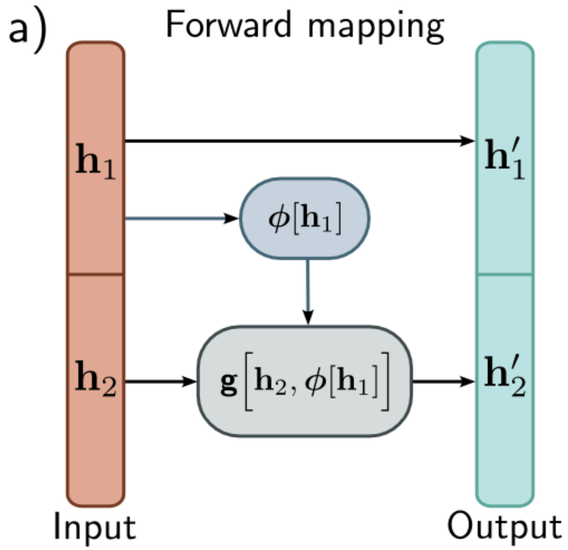
$$\mathrm{f}[h, \boldsymbol{\phi}] = \left(\sum_{k=1}^{b-1} \phi_k\right) + (hK - b)\phi_b$$



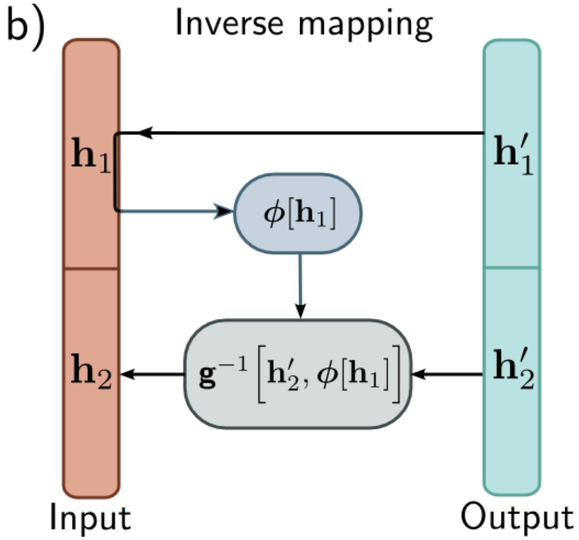


Coupling flows

- *Couplings flows operate as follows*
  - *Split input h in two vectors h_1 and h_2*
    - *h_1 is passed to output as it is*
    - *h_1 is used to compute some vector of parameters $\phi[h_1]$*
    - *$\phi[h_1]$ are used as the parameters of an elemetwise (or any other invertible) flow that acts on h_2*
  - *Jakobian is triangular (h'_2 depends on h_1, but h'_1 does not depend on on h_2)*
  - *One usually shuffles the elements in h before splitting it into h_1 and h_2 at every layer, so that going deep in the chain every input lands in one of the h_2 (so that everything is transformed)*



$$\mathbf{h}'_1 = \mathbf{h}_1$$
$$\mathbf{h}'_2 = \mathbf{g}\left[\mathbf{h}_2, \phi[\mathbf{h}_1]\right]$$

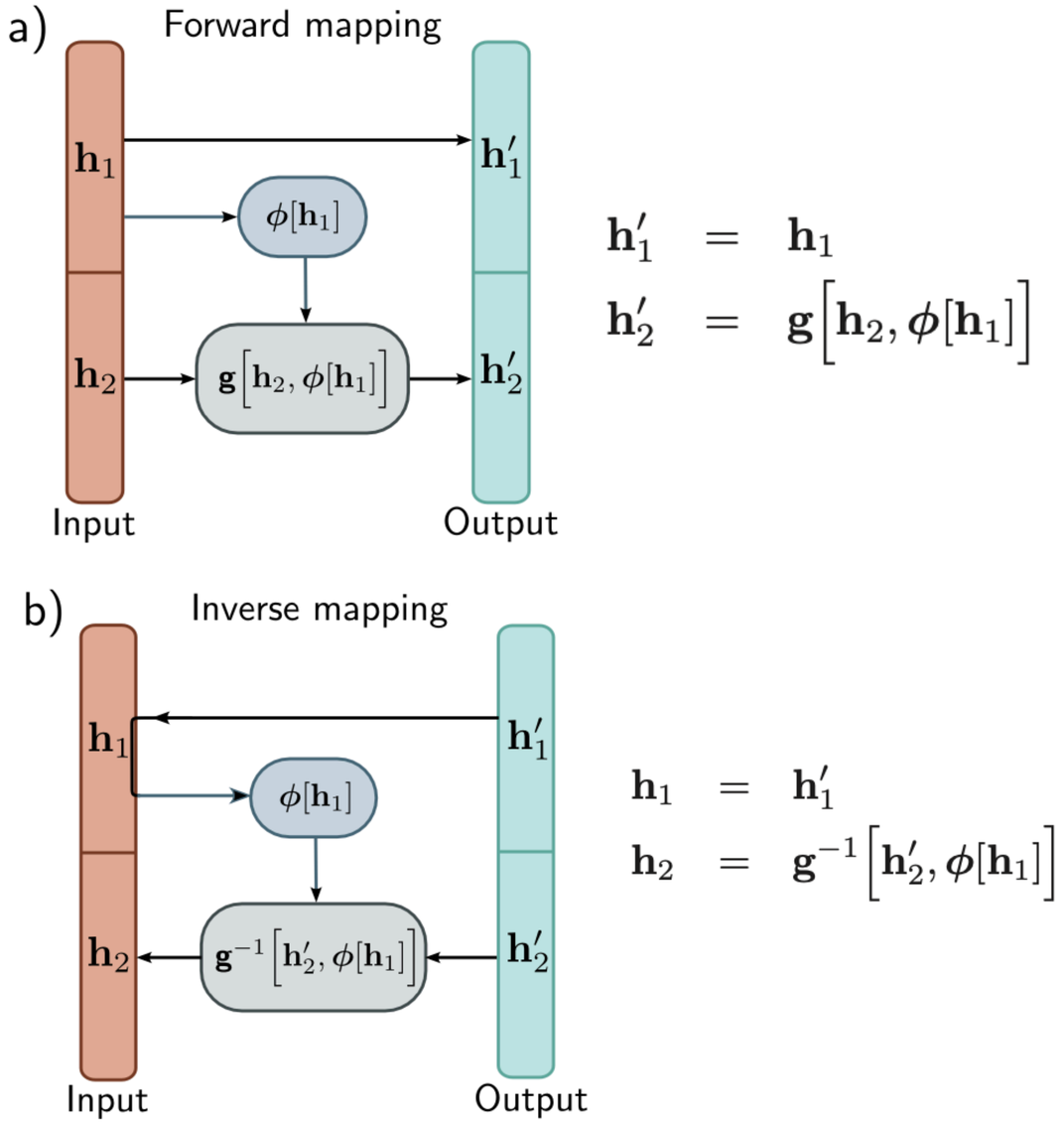


$$\mathbf{h}_1 = \mathbf{h}'_1$$
$$\mathbf{h}_2 = \mathbf{g}^{-1}\left[\mathbf{h}'_2, \phi[\mathbf{h}_1]\right]$$

RealNVP

- *Real-valued Non-Volume Preserving (or RealNVP) are special kinds of flows in which*
  - *a mask is used to shuffle the inputs*
  - *The affine transformation g is then obtained*
    - *Computing the output of two dense networks*
      - *$s(h_1)$ is the scale to apply on h_2*
      - *$t(h_1)$ is the shift to apply to h_2*
    - *Computing h'_2 as*

$$h'_2 = h_2 \cdot e^{s(h_1)} + t(h_1)$$

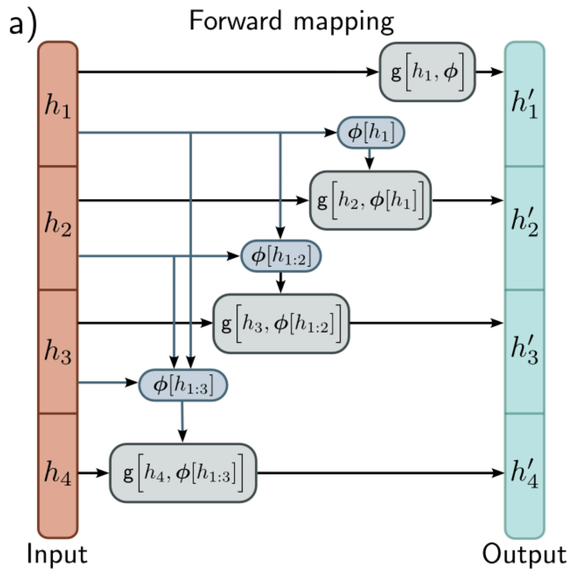


Autoregressive flows

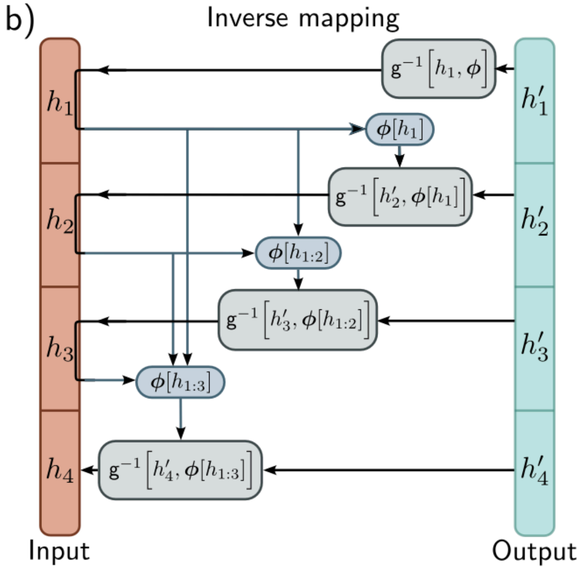
- *Generalize coupling flows to enhance complexity and expressivity*
  - *Each input is a separate block*
  - *Each block is transformed according to a transformer function $g[h_i, \phi]$ (not related to transformer networks!!!), depending on conditioners $\phi = [\phi_{1:i}]$*

$$h'_d = \mathrm{g}\Big[h_d, \boldsymbol{\phi}[\mathbf{h}_{1:d-1}]\Big]$$

- *Notice: since the computation of h_d depends on $h_{1:d-1} = [h_1, h_2, \ldots, h_{d-1}]$, it cannot proceed in parallel (which slows down the inference)*
- *For similar reasons, the inversion has to be done sequentially*



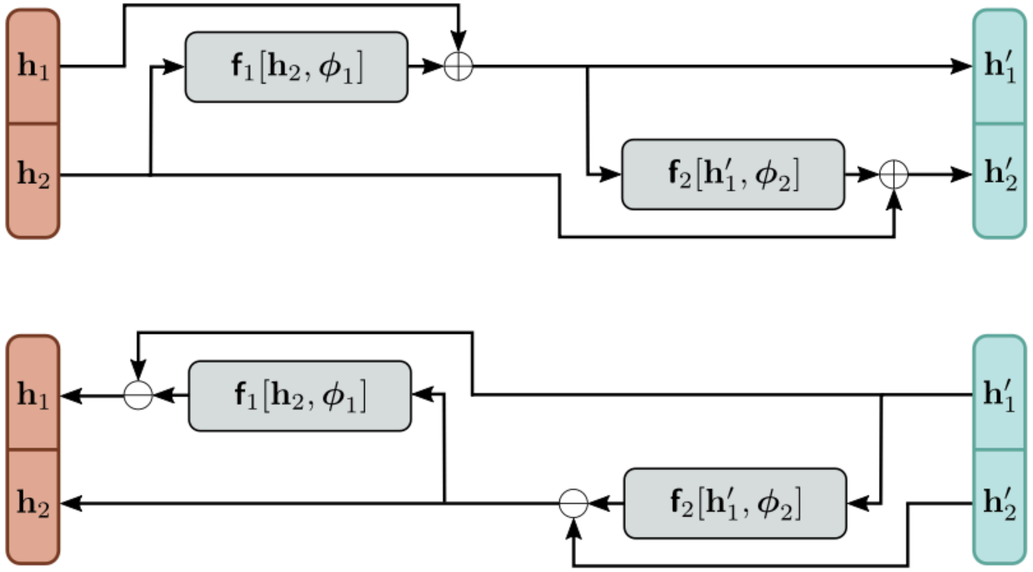
$$\begin{aligned} h'_1 &= \mathrm{g}\Big[h_1, \boldsymbol{\phi}\Big] \\ h'_2 &= \mathrm{g}\Big[h_2, \boldsymbol{\phi}[h_1]\Big] \\ h'_3 &= \mathrm{g}\Big[h_3, \boldsymbol{\phi}[h_{1:2}]\Big] \\ h'_4 &= \mathrm{g}\Big[h_4, \boldsymbol{\phi}[h_{1:3}]\Big] \end{aligned}$$



$$\begin{aligned} h_1 &= \mathrm{g}^{-1}\Big[h'_1, \boldsymbol{\phi}\Big] \\ h_2 &= \mathrm{g}^{-1}\Big[h'_2, \boldsymbol{\phi}[h_1]\Big] \\ h_3 &= \mathrm{g}^{-1}\Big[h'_3, \boldsymbol{\phi}[h_{1:2}]\Big] \\ h_4 &= \mathrm{g}^{-1}\Big[h'_4, \boldsymbol{\phi}[h_{1:3}]\Big] \end{aligned}$$

Residual flows

- *Architecture inspired from ResNet CNNs*
  - *as for couplings flows, split* h *in* $[h_1, h_2]$
  - *Both are transformed (sequentially)*
    - h_2 *determines* $h_1' = g[h_2, \phi_1] + h_1$ *(adding* h_1 *through a skip connection is what comes from the ResNet inspiration)*
    - h_1' *goes to outputs and it also determines* $h_2' = g[h_1, \phi_2] + h_2$

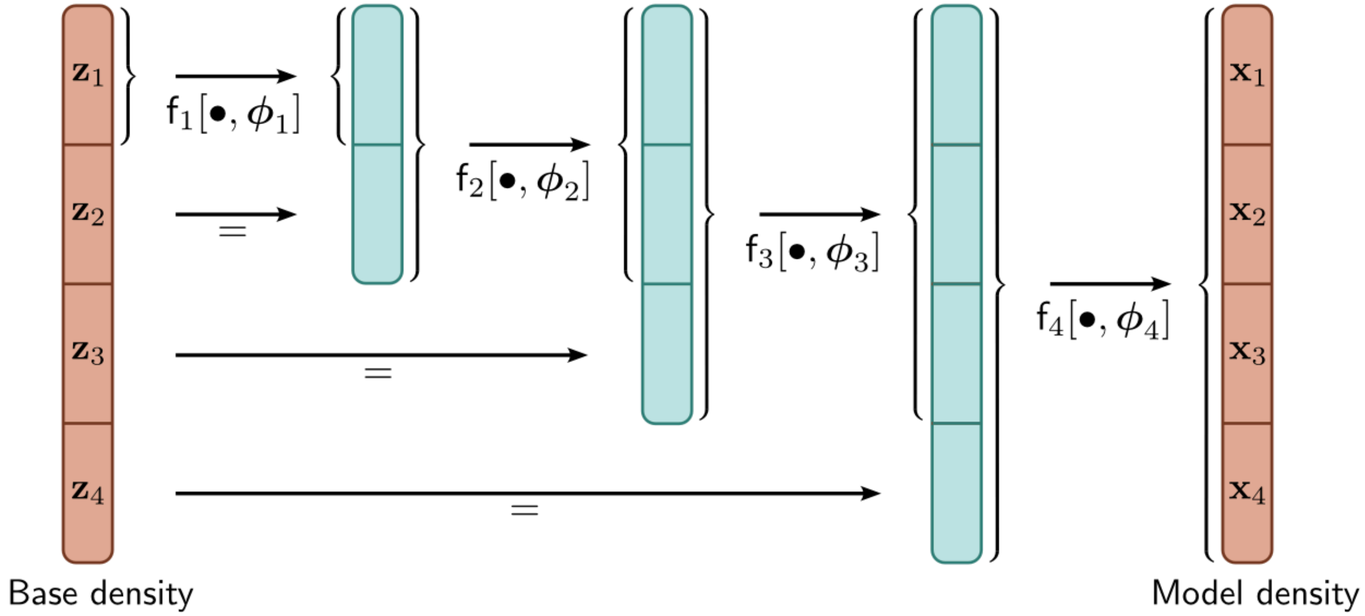


$$\begin{aligned} \mathbf{h}_1' &= \mathbf{h}_1 + \mathbf{f}_1[\mathbf{h}_2, \phi_1] \\ \mathbf{h}_2' &= \mathbf{h}_2 + \mathbf{f}_2[\mathbf{h}_1', \phi_2]; \end{aligned}$$

$$\begin{aligned} \mathbf{h}_2 &= \mathbf{h}_2' - \mathbf{f}_2[\mathbf{h}_1', \phi_2] \\ \mathbf{h}_1 &= \mathbf{h}_1' - \mathbf{f}_1[\mathbf{h}_2, \phi_1] \end{aligned}$$

Multi-scale flows

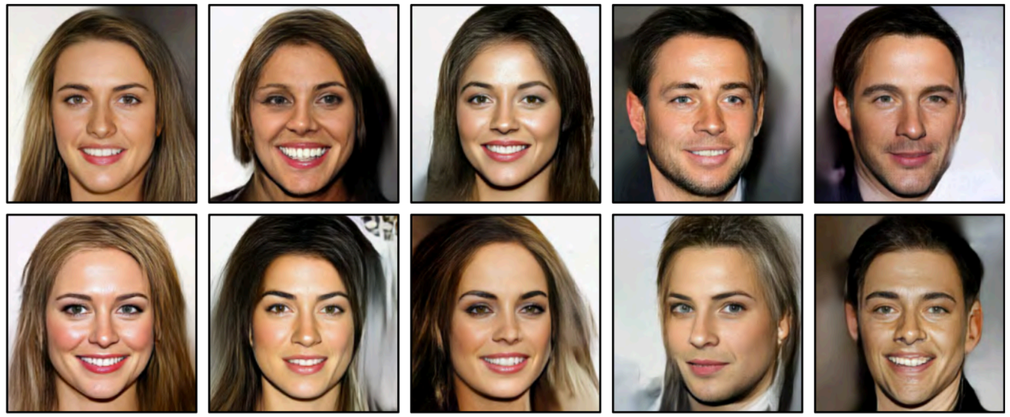
- *To be invertible, flows need to have outputs of the same dimension as inputs*
- *On the other hand, some part of the computation might only depend on a subset (in which case, carrying around the entire input when only a faction is needed would be inefficient)*
- *In multi-scale flows, this is achieved factorizing the computation and introducing additional elements progressively*



Applications

- *Modeling Densities*: unlike other generative models, one can learn the likelihood of the sample
  - Why GANs and VAEs can't?
- *Synthesis*: can generate images
  - GLOW is an example of NormalizingFlow generating faces.
    - It starts from images (256x256x3)
    - It uses coupling flows, in which the transformation imposed on h_2 depends on the position and is learned as a 2D CNN applied on h_1
- *Approximating other density models*: they can lean to approximate other functions that might be more complicated to sample from. This will be more clear when we will discuss knowledge distillation

GLOW can generate faces



One can interpolate between faces, following a linear trajectory in the latent space and applying GLOW to each point