

# Deep Learning for Data Science

## DS 542

Lecture 22  
Normalizing Flows



# Last Time

## Unsupervised Learning

- Taxonomy
- Generative models
- Quantifying performance
- Variational autoencoders

# This Time

## Generative Models

- Normalizing Flows
  - Key design change is invertibility of each layer
  - Enables efficient probability computations

# Do we have good models?

	GANs	VAEs	Flows	Diffusion
Efficient sampling	✓	✓	✓	✗
High quality	✓	✗	✗	✓
Coverage	✗	?	?	?
Well-behaved latent space	✓	✓	✓	✗
Interpretable latent space	?	?	?	✗
Efficient likelihood	n/a	✗	✓	✗

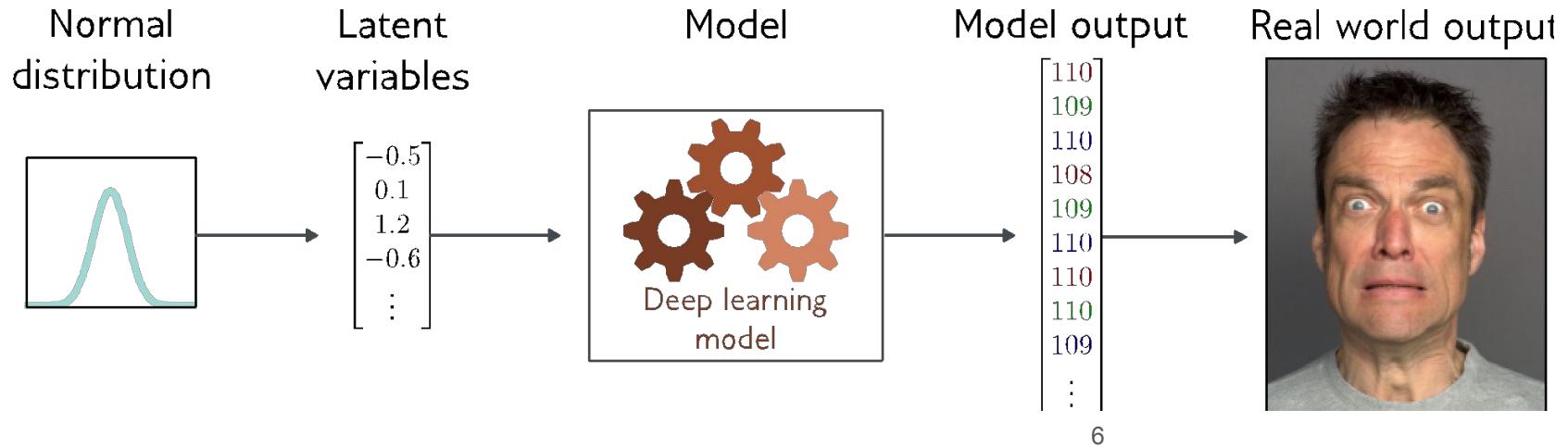
# Didn't Variational Autoencoders have Sample Probabilities?

Yes, but can you solve this efficiently?

$$Pr(\mathbf{x}_i | \phi) = \int \text{Norm}_{\mathbf{x}_i} [\mathbf{f}(\mathbf{z}, \phi), \sigma^2 \mathbf{I}] \text{Norm}_{\mathbf{z}}[\mathbf{0}, \mathbf{I}] d\mathbf{z}$$

This can be approximated by sampling z but slow...

# Latent variable models



Latent variable models map a random “latent” variable to create a new data sample

# Latent Variable Models

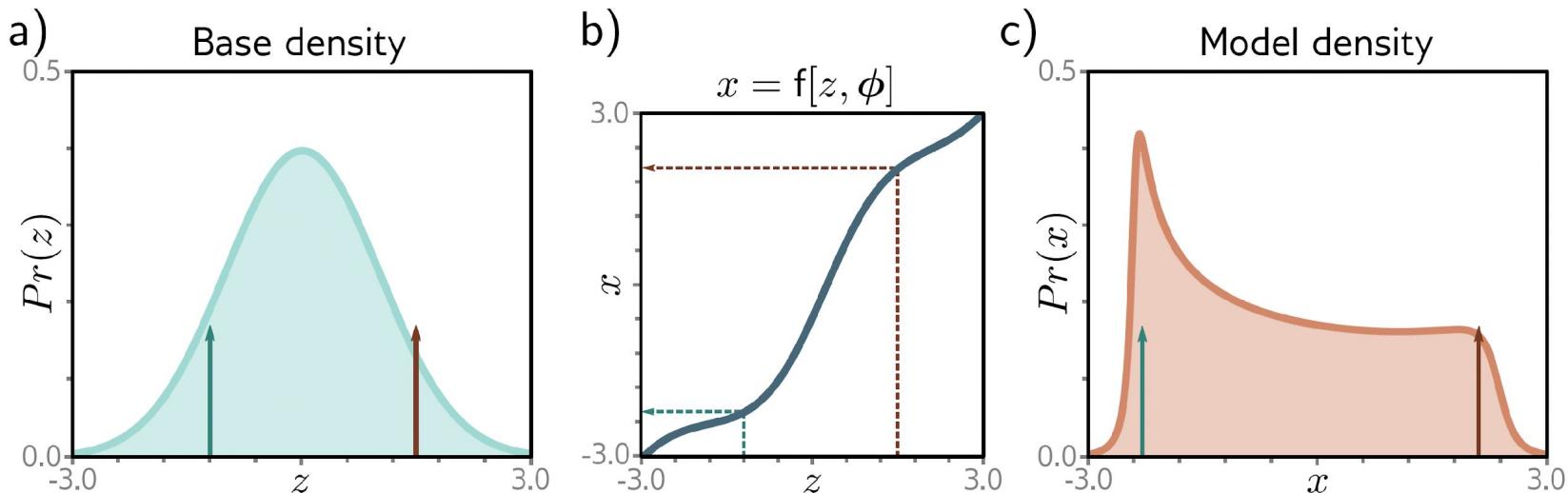
Informally speaking, different levels of latent variables...

- Latent variable directly determines observations
  - e.g.  $x = f(z)$
- Latent variable determines distribution of observations
  - e.g.  $x \approx \text{Norm}[f_\mu(z), f_{\sigma^2}(z)]$
- These levels aren't really different -
  - An extremely tight distribution  $\sim$  a fixed prediction
  - A fixed prediction + noise  $\sim$  a distribution

# Normalizing Flows

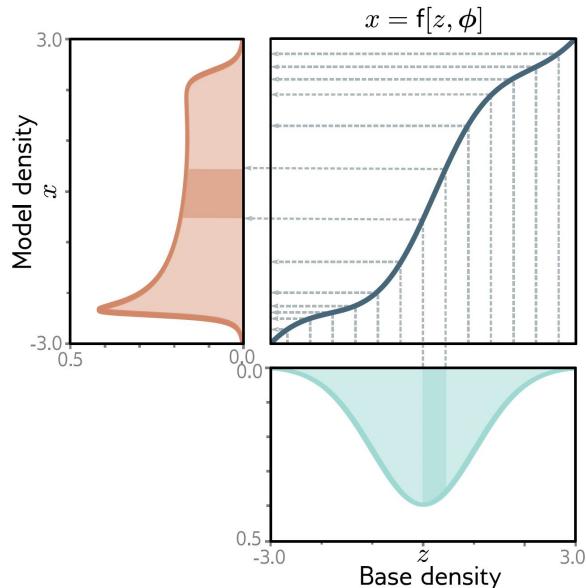
- Will start from a latent variable with a known distribution.
- Will map directly from latent variable to sample value.
  - $\mathbf{x} = \mathbf{f}(\mathbf{z})$
- Key change:  $\mathbf{f}$  will be invertible.
  - This also means that  $\mathbf{z}$  and  $\mathbf{x}$  have the same dimension.

# One Dimension - Mapping from Latent to Sample



**Figure 16.1** 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  $x = f[z, \phi]$  to a new variable  $x$ , which c) has a new distribution. 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  $f[z, \phi]$  as shown by dotted arrows in panel (b) to generate the values of  $x$ , which are indicated as arrows in panel (c).

# One Dimension - Mapping from Latent to Sample



**Figure 16.2** 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.

# One Dimension - Easy Mode

Easier trick (not in the book)

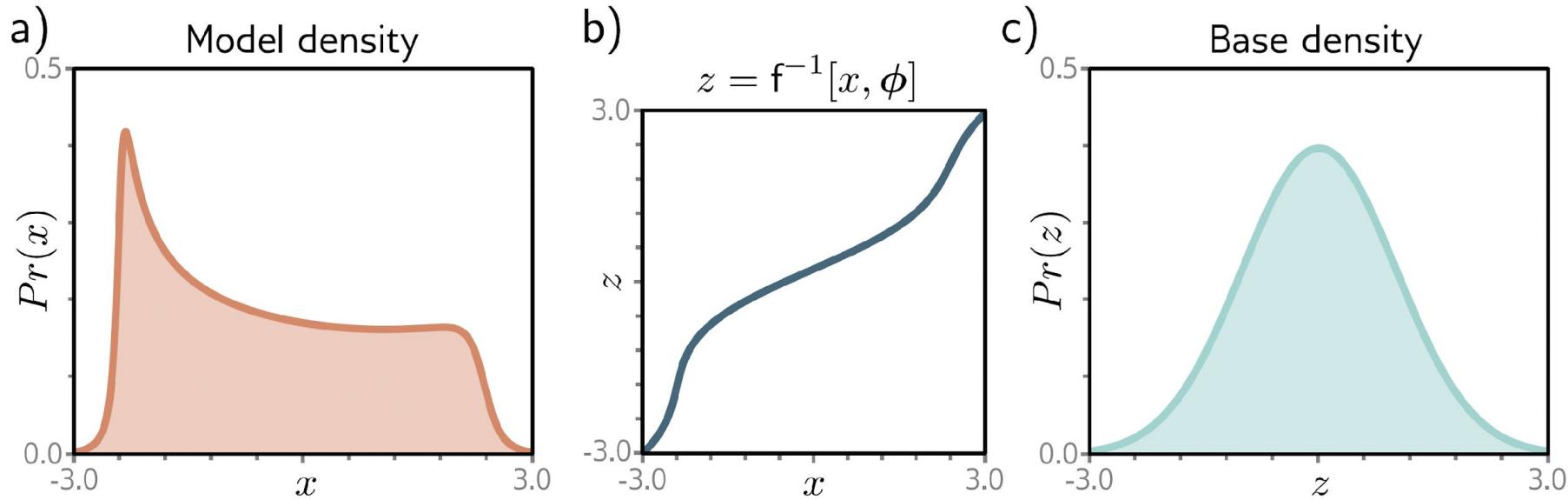
- Use cumulative distribution functions!

$$x = \text{cdf}_x^{-1}(\text{cdf}_z(z))$$

- This is similar to the sampling method

$$x = \text{cdf}_x^{-1}(\text{Uniform}(0,1))$$

# One Dimension - Inverting



**Figure 16.3** 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 (see equation 16.1).

## One Dimension - Inverting Easy Mode

If still one dimensional,

$$x = \text{cdf}_x^{-1}(\text{cdf}_z(z))$$

Then invert to get

$$z = \text{cdf}_z^{-1}(\text{cdf}_x(x))$$

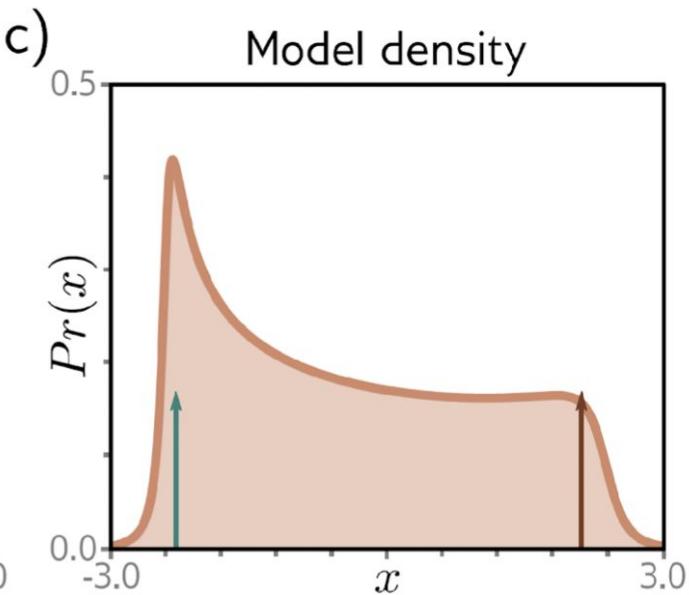
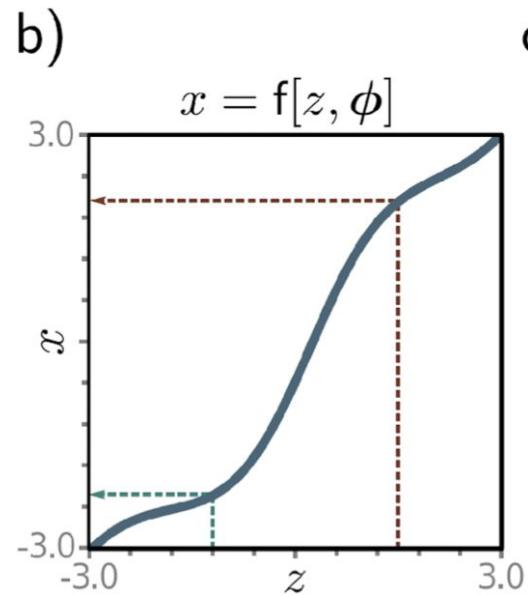
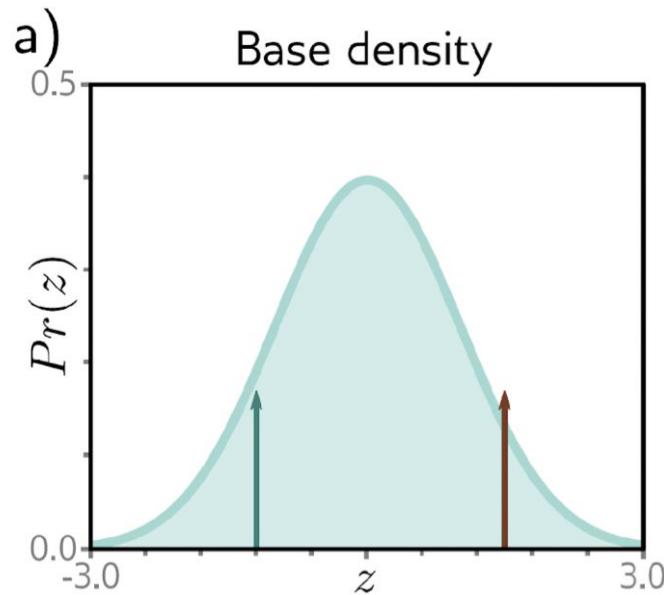
# One Dimension - Mapping Probabilities

Simple formula for sample probability density in terms of latent probability density and derivative of mapping.

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

- If mapping changes quickly, probability is spread out more.
- If mapping changes slowly, probability is more concentrated.

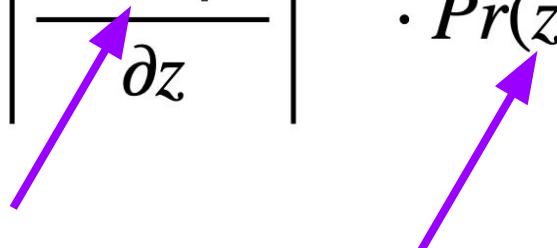
# One Dimension - Mapping Probabilities



# Requirement for Computing Sample Probability

We need to know  $z$ !

- So  $f$  must be invertible.
- Easy to handle in 1D.
  - Just use cdf's...
- Will be harder with more dimensions.

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


## Aside: Terminology

- The forward mapping  $\mathbf{x} = \mathbf{f}[\mathbf{z}, \phi]$  is often called the **generative** direction.
- The inverse mapping  $\mathbf{z} = \mathbf{f}^{-1}[\mathbf{x}, \phi]$  is often called the **normalizing** direction since it maps the complicated  $\mathbf{x}$  distribution to the relatively simple  $\mathbf{z}$  distribution... which we usually pick to be normal.

# Why Normal Latents?

Why do we keep using normal distributions for latent variables?

- They are defined everywhere, so any value works.
  - So they have gradients everywhere.
  - But those gradients induce a bias towards more central latent values.
  - And outliers are obvious.
- 
- Variational autoencoders used these properties explicitly a couple times.
  - Similar benefits here.

# General Case (more than one dimension)

Still using  $\mathbf{x} = \mathbf{f}[\mathbf{z}, \phi]$  but now using deep neural networks.

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

- “Same formulas” but with bold for vector notation instead of scalars.
- In the sample probability formula,
  - Partial derivatives are the Jacobian matrix.
  - And the magnitude is the determinant of that Jacobian matrix.

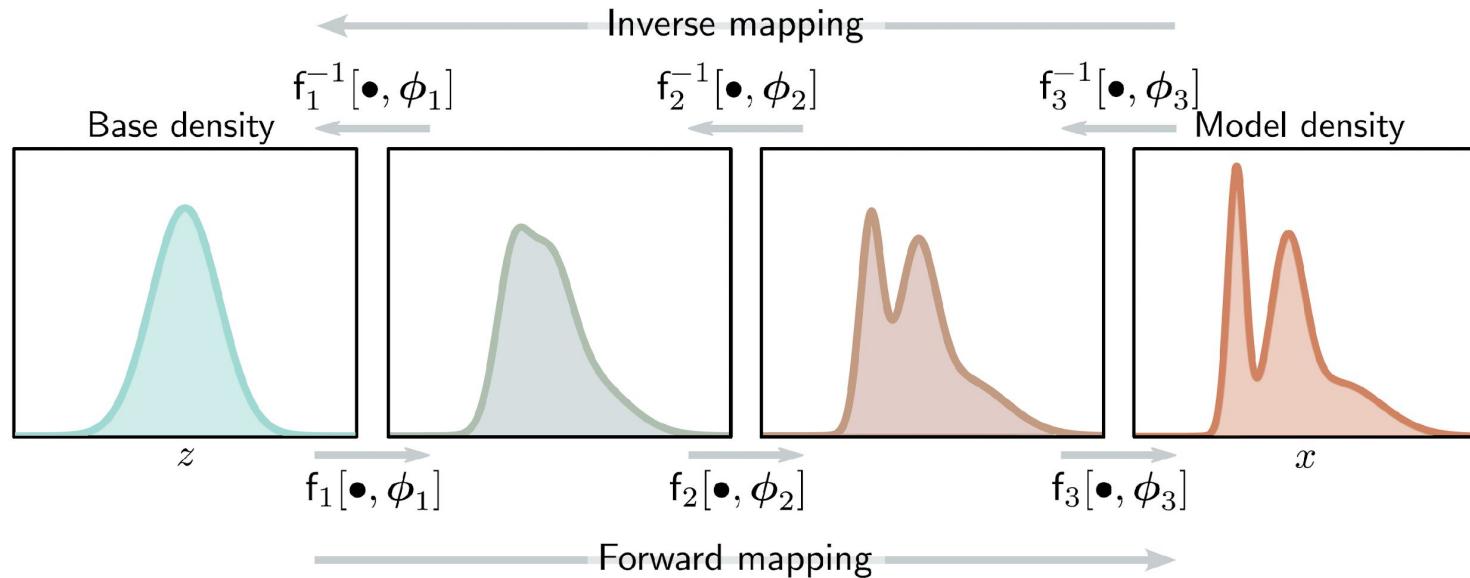
# Why Determinant?

- When I learned linear algebra,
  - $\det(A)$  = sum over column permutations of the product of the diagonal plus a power of -1...
  - That is pretty irrelevant now.
- More relevant identity
  - $\det(A)$  = product of eigenvalues

# Determinants and relative density changes...

- Eigenvectors form an orthonormal basis.
- Looking at eigenvectors of the gradients, so they span the sample space.
- So the eigenvalues tell us about relative rates of gradient change.
- And the determinant tells about relative volume rates  
between latent and sample space...

# Forward and Inverse Mappings



**Figure 16.4** Forward and inverse mappings for a deep neural network. The base density (left) is gradually transformed by the network layers  $f_1[\bullet, \phi_1], f_2[\bullet, \phi_2], \dots$  to create the model density. Each layer is invertible, and we can equivalently think of the inverse of the layers as gradually transforming (or “flowing”) the model density back to the base density.

# General Case

Unwinding the layers of the neural network,

$$\mathbf{x} = \mathbf{f}[\mathbf{z}, \phi] = \mathbf{f}_K \left[ \mathbf{f}_{K-1} \left[ \dots \mathbf{f}_2 \left[ \mathbf{f}_1[\mathbf{z}, \phi_1], \phi_2 \right], \dots \phi_{K-1} \right], \phi_K \right]$$

$$\mathbf{z} = \mathbf{f}^{-1}[\mathbf{x}, \phi] = \mathbf{f}_1^{-1} \left[ \mathbf{f}_2^{-1} \left[ \dots \mathbf{f}_{K-1}^{-1} \left[ \mathbf{f}_K^{-1}[\mathbf{x}, \phi_K], \phi_{K-1} \right], \dots \phi_2 \right], \phi_1 \right]$$

Again, the usual choice for  $\Pr(z)$  is a normal distribution, leading to the name “normalizing flows” for this inverting process.

## General Case - Jacobian

The Jacobian can similarly be computed from individual layer Jacobians...

$$\frac{\partial \mathbf{f}[\mathbf{z}, \phi]}{\partial \mathbf{z}} = \frac{\partial \mathbf{f}_K[\mathbf{f}_{K-1}, \phi_K]}{\partial \mathbf{f}_{K-1}} \cdot \frac{\partial \mathbf{f}_{K-1}[\mathbf{f}_{K-2}, \phi_{K-1}]}{\partial \mathbf{f}_{K-2}} \cdots \frac{\partial \mathbf{f}_2[\mathbf{f}_1, \phi_2]}{\partial \mathbf{f}_1} \cdot \frac{\partial \mathbf{f}_1[\mathbf{z}, \phi_1]}{\partial \mathbf{z}}$$

If any of them is zero, then the whole Jacobian is zero.

Also the case for invertibility.

# Layer Wish List

- Expressiveness - can they combine to express an arbitrary density mapping?
- Invertibility - otherwise, not normalizing flows.
- Efficiency
  - Inverting each layer should be efficient.
  - Computing the determinant of the Jacobian should also be efficient (either direction works).

# Invertible Network Layers

Two easy choices for invertible network layers or “flows”

- Linear flows
- Elementwise flows

Both are easy to invert, but not expressive enough.

- Will use these as building blocks.

# Normalizing Flows - Linear Flows

Really just a linear layer without an activation function.

$$\mathbf{f}[\mathbf{h}] = \beta + \Omega \mathbf{h}$$

Cost of inverting grows with cube of dimension, so may parameterize it as an LU decomposition to make inversion cheaper.

$$\Omega = PL(U + D)$$

Linear functions applied to multivariate normal distributions give more multivariate normal distributions.

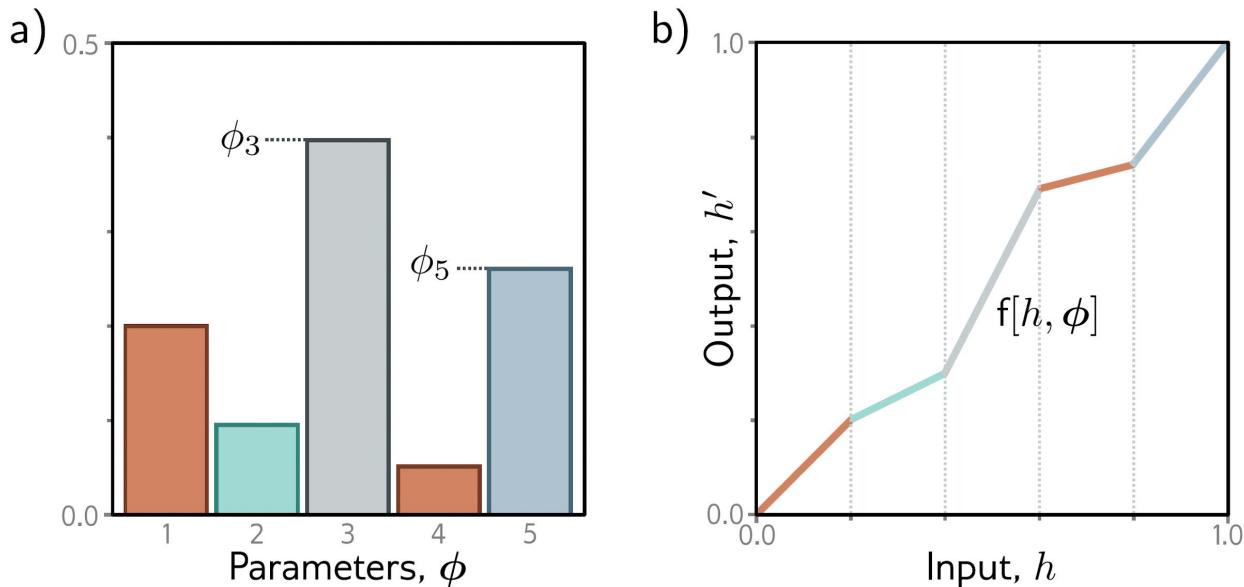
# Elementwise Flows

Idea: Apply an invertible nonlinear function to each element.

$$\mathbf{f}[\mathbf{h}] = [f[h_1, \phi], f[h_2, \phi], \dots, f[h_D, \phi]]^T$$

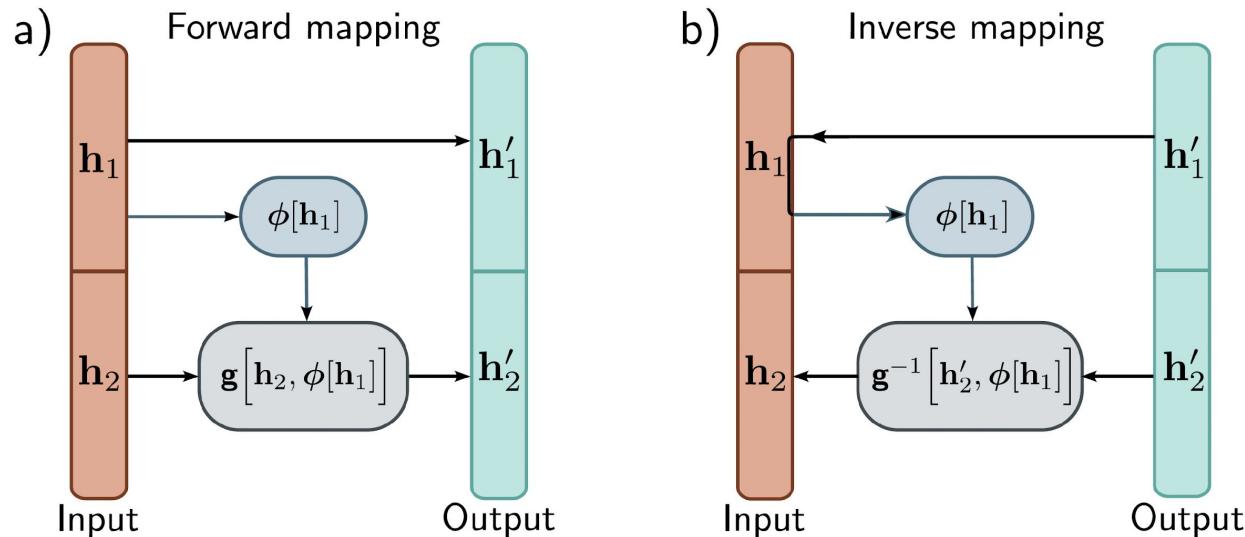
- Can use different functions for each element.
- Limitation is that different dimensions do not interact.

# Elementwise Flows



**Figure 16.5** 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,  $\phi_k$ . a) If these parameters are positive and sum to one, then b) the function will be invertible and map to the output domain  $h' \in [0, 1]$ .

# Coupling Flows

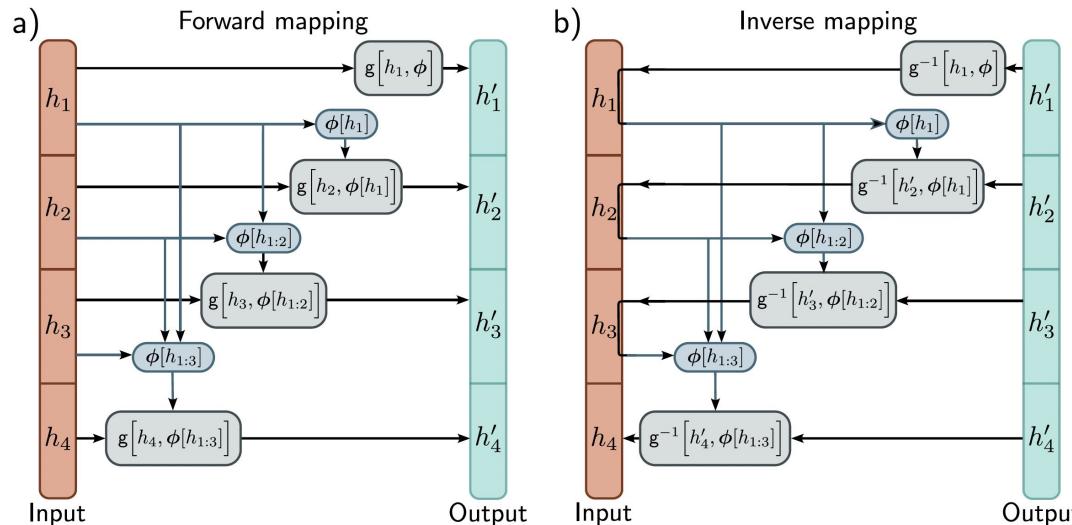


**Figure 16.6** Coupling flows. a) The input (orange vector) is divided into  $\mathbf{h}_1$  and  $\mathbf{h}_2$ . The first part  $\mathbf{h}'_1$  of the output (cyan vector) is a copy of  $\mathbf{h}_1$ . The output  $\mathbf{h}'_2$  is created by applying an invertible transformation  $\mathbf{g}[\bullet, \phi]$  to  $\mathbf{h}_2$ , where the parameters  $\phi$  are themselves a (not necessarily invertible) function of  $\mathbf{h}_1$ . b) In the inverse mapping,  $\mathbf{h}_1 = \mathbf{h}'_1$ . This allows us to calculate the parameters  $\phi[\mathbf{h}_1]$  and then apply the inverse  $\mathbf{g}^{-1}[\mathbf{h}'_2, \phi]$  to retrieve  $\mathbf{h}_2$ .

# Coupling Flows

- Half of vector is just copied.
- Other half can change.
- Use fixed permutation matrices to vary which can change each layer.
  - Permutation matrices tend to be hard to learn.

# Autoregressive Flows

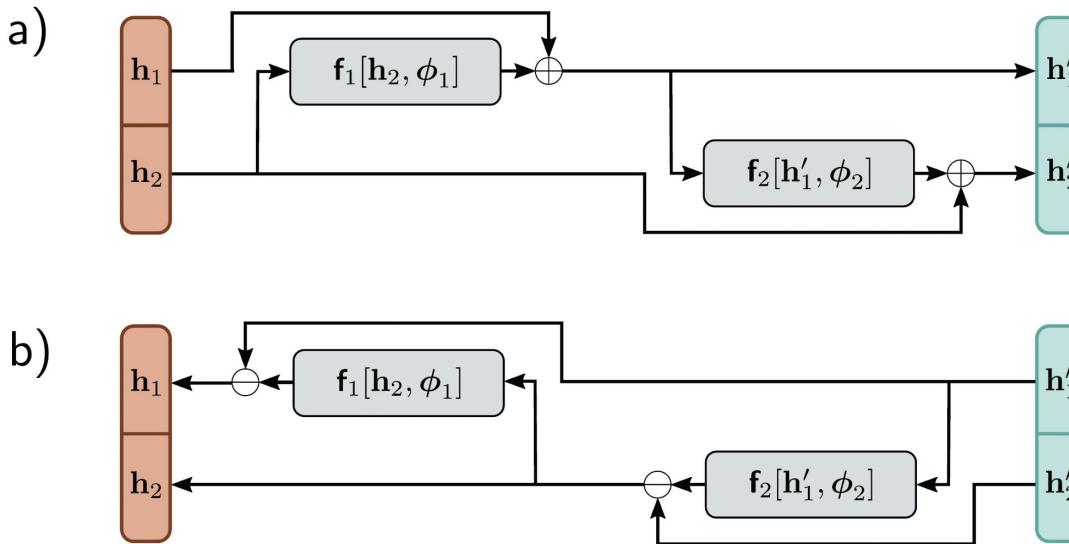


**Figure 16.7** Autoregressive flows. The input  $\mathbf{h}$  (orange column) and output  $\mathbf{h}'$  (cyan column) are split into their constituent dimensions (here four dimensions). a) Output  $h'_1$  is an invertible transformation of input  $h_1$ . Output  $h'_2$  is an invertible function of 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.

# Inverse Autoregressive Flows

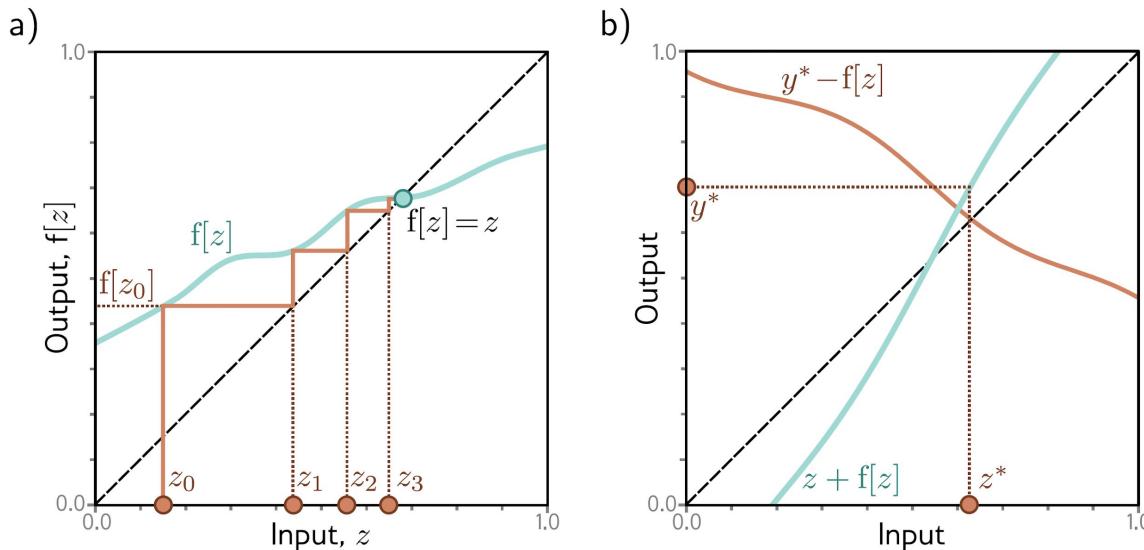
- Inverting can be done in the reverse order of computation.
- But this means more elaborate (deeper) flows are slower to invert.
- If we just want sample probabilities,
  - Build backwards so forward is slow and inverse is fast!
  - Called an “inverse autoregressive flow”.
- Another approach is to train another network to learn the inverse.
  - Less precise, but can be a lot faster.
  - Easy to generate samples by known (picked) latent distribution.

# Residual Flows



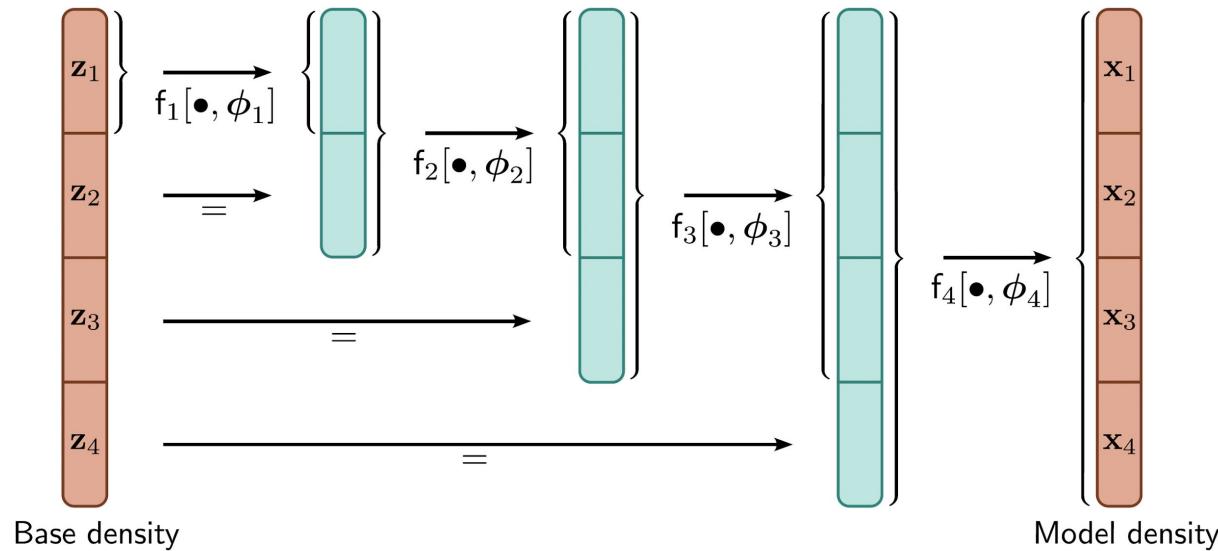
**Figure 16.8** Residual flows. a) An invertible function is computed by splitting the input into  $\mathbf{h}_1$  and  $\mathbf{h}_2$  and creating two residual layers. In the first,  $\mathbf{h}_2$  is processed and  $\mathbf{h}_1$  is added. In the second, the result is processed, and  $\mathbf{h}_2$  is added. b) In the reverse mechanism the functions are computed in the opposite order, and the addition operation becomes subtraction.

# Contraction Mappings



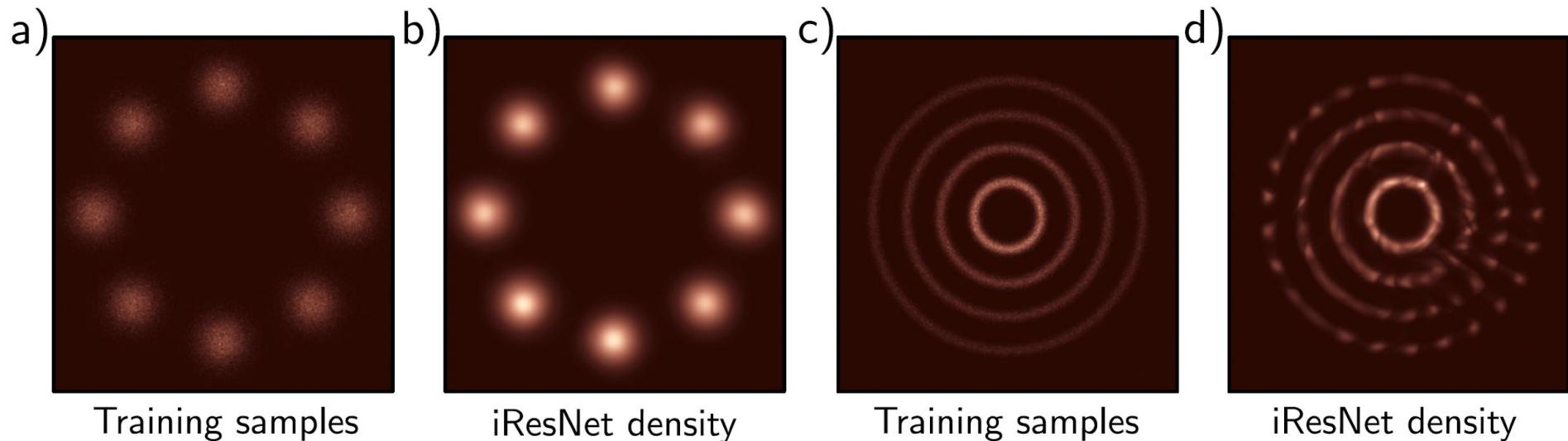
**Figure 16.9** Contraction mappings. If a function has an absolute slope of less than one everywhere, iterating the function converges to a fixed point  $f[z] = z$ . a) Starting at  $z_0$ , we evaluate  $z_1 = f[z_0]$ . We then pass  $z_1$  back into the function and iterate. Eventually, the process converges to the point where  $f[z] = z$  (i.e., where the function crosses the dashed diagonal identity line). b) This can be used to invert equations of the form  $y = z + f[z]$  for a value  $y^*$  by noticing that the fixed point of  $y^* - f[z]$  (where the orange line crosses the dashed identity line) is at the same position as where  $y^* = z + f[z]$ .

# Multiscale Flows



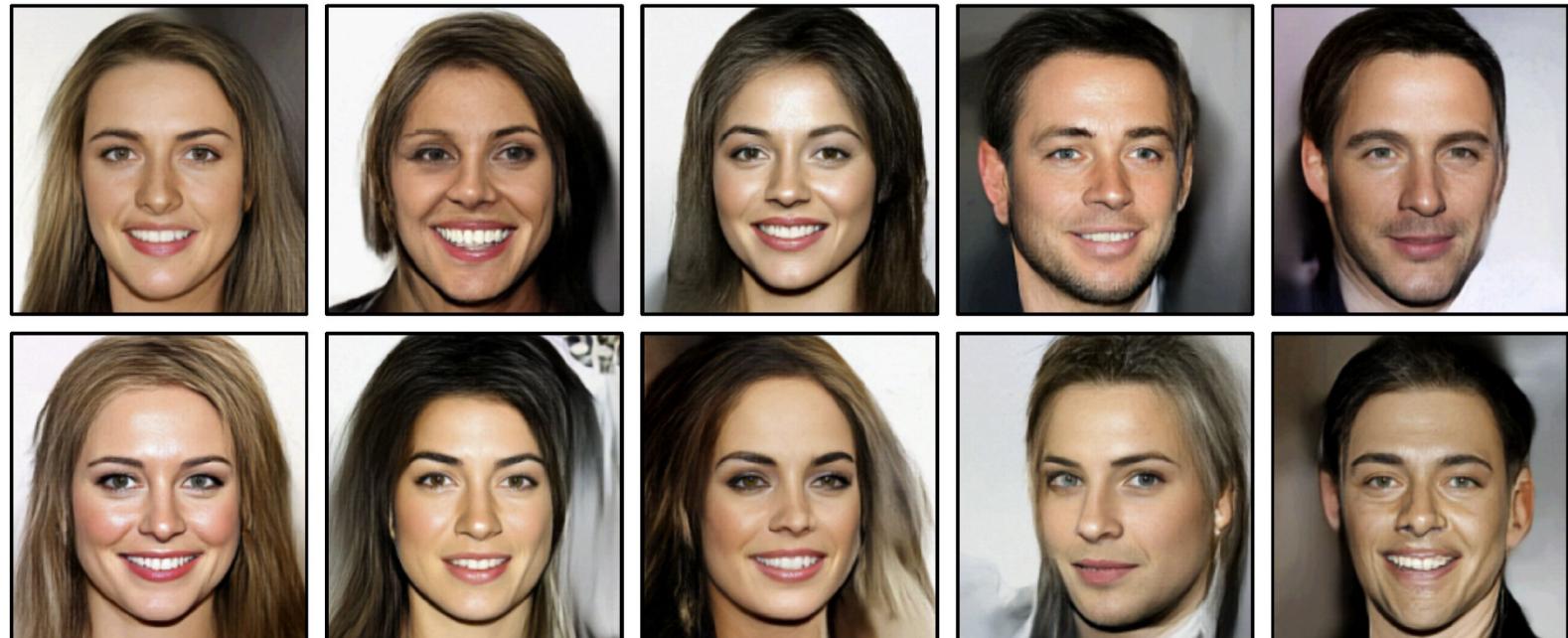
**Figure 16.10** Multiscale flows. The latent space  $\mathbf{z}$  must be the same size as the model density in normalizing flows. However, it can be partitioned into several components, which can be gradually introduced at different layers. This makes both density estimation and sampling faster. For the inverse process, the black arrows are reversed, and the last part of each block skips the remaining processing. For example,  $f_3^{-1}[\bullet, \phi_3]$  only operates on the first three blocks, and the fourth block becomes  $\mathbf{z}_4$  and is assessed against the base density.

# Modeling Densities (only normalizing flows can do)



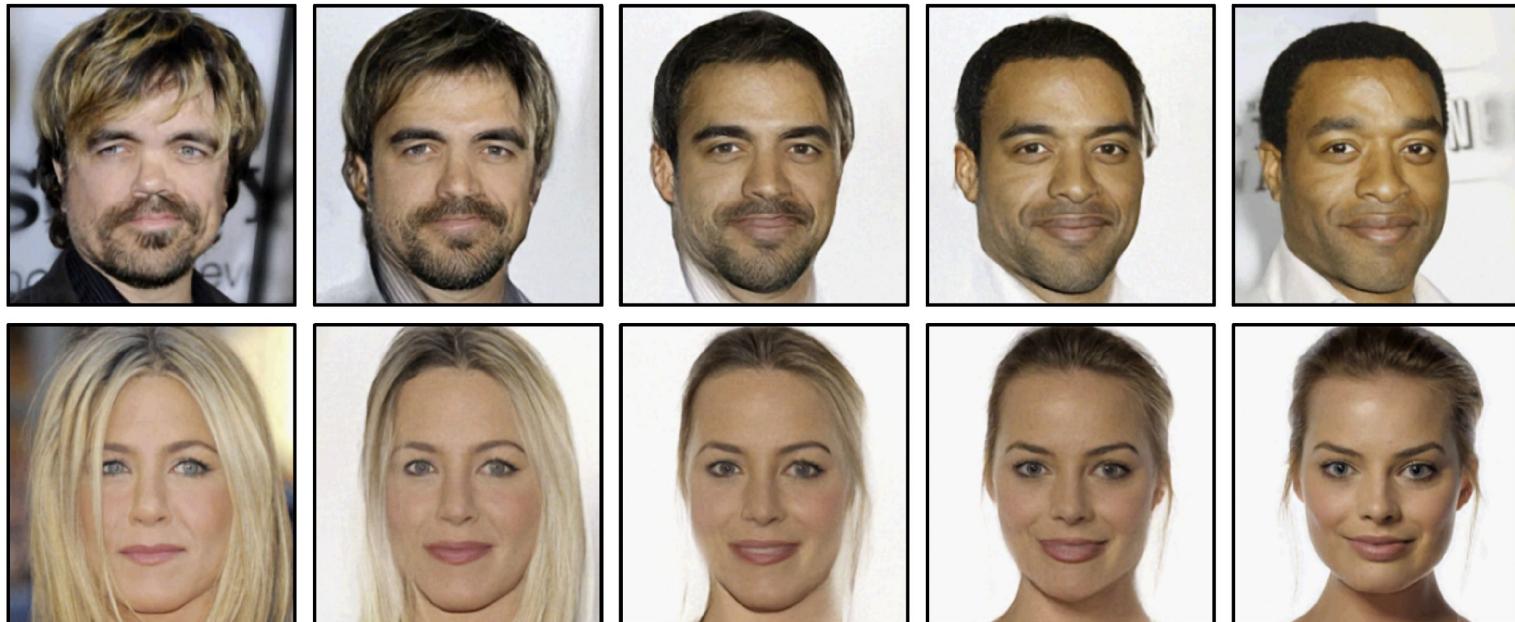
**Figure 16.11** Modeling densities. a) Toy 2D data samples. b) Modeled density using iResNet. c–d) Second example. Adapted from Behrmann et al. (2019)

## Synthesis (note artifacts)



**Figure 16.12** 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

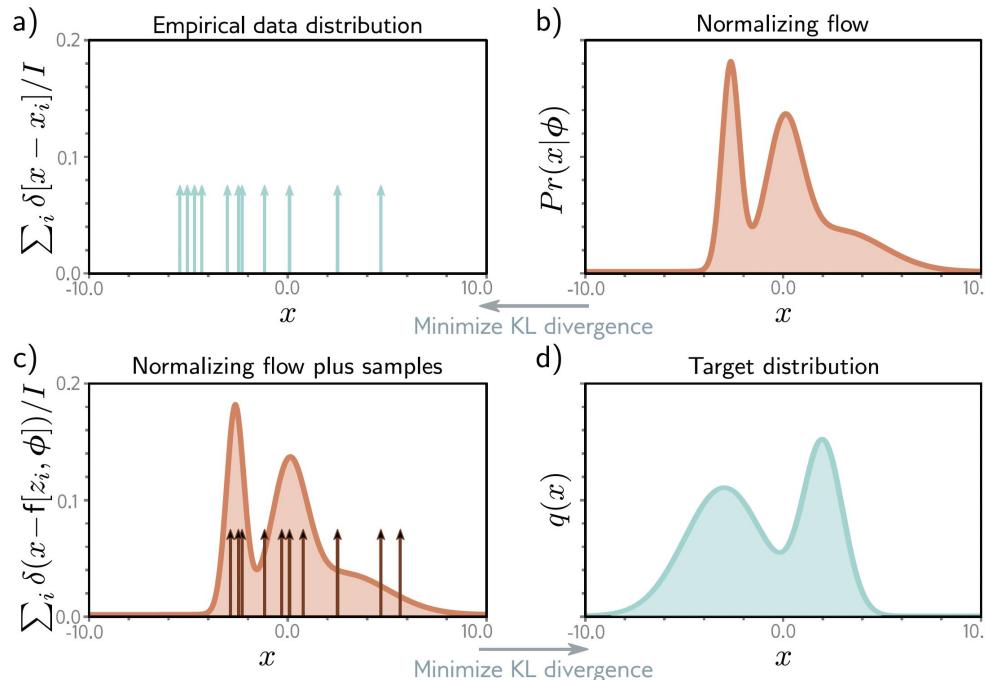


**Figure 16.13** Interpolation using GLOW model. The left and right images are real people. The intermediate images were computed by projecting the real images to the latent space, interpolating, and then projecting the interpolated points back to image space. Adapted from Kingma & Dhariwal (2018).

# Glow Notes

- Used progressive modeling of different resolutions.
  - Some of the flow types we saw make this easy.
  - Also added noise to avoid overfitting discrete pixel values.
- They were not confident about outliers.
  - Sampled from square of probability distribution, so more concentrated in center.
- There is more than one model called GLOW.
  - The one I thought of seeing this name has GAN quality.

# Approximating Other Density Models (including VAE)



**Figure 16.14** 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 (section 5.7). c) Alternatively, we can modify the flow parameters  $\phi$  to minimize the KL divergence from the flow samples  $x_i = f[z_i, \phi]$  to d) a target density.

# Normalizing Flows - The Cost of Invertibility

Representational aspects of depth  
and conditioning in normalizing flows

Frederic Koehler\*, Viraj Mehta†, Andrej Risteski‡

October 6, 2020

“For general invertible architectures, we prove that invertibility comes at a cost in terms of depth: we show examples where a **much deeper normalizing flow model may need to be used to match the performance of a non-invertible generator.**”

# Rest of the Semester

- Diffusion Models (11/20)
- Neural Fields (11/25)
- Thanksgiving break (11/27)
- Reinforcement learning (12/2)
- Project presentations (12/4)
- Project presentations (12/9)

# Feedback?

