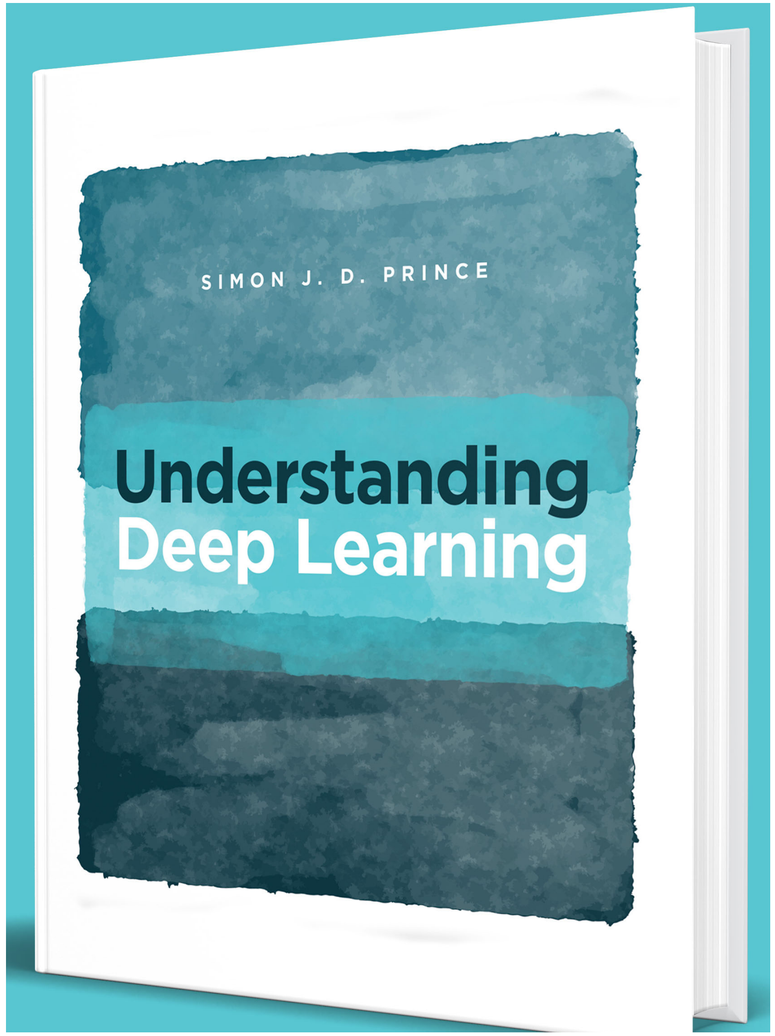
Lecture 9: Generative Models

Program for Today

Date	Topic	Tutorial
Sep 17	Intro & class description	Linear Algebra in a nutshell + prob and stat
Sep 24	Basic of machine learning + Dense NN	*Basic jupyter + DNN on mnist (give jet dnn as homework)*
Oct 1	Convolustional NN	*Convolutional NNs with MNIST*
Oct 8	Training in practice: regularization, optimization, etc	*Practical methodology*
Oct 15	*Google tutorial*	
Oct 22	Recurrent NN	*Hands-on exercise*
Oct 29	Graph NNs	*Tutorial on Graph NNs*
Nov 5	Unsupervised learning and anomaly detection	*Autoencoders with MNIST*
→ **Nov 12**	Generative models: GANs, VAEs, etc	Normalizing flows
Nov 19	Transformers	
Nov 26	Network compression (pruning, quantization, Knoweldge Distillation)	
Dec 3	Tutorial on hls4ml/qkeras	
Dec 10	Quantum Machine Learning	
Dec 17	Q/A Prior to exam	

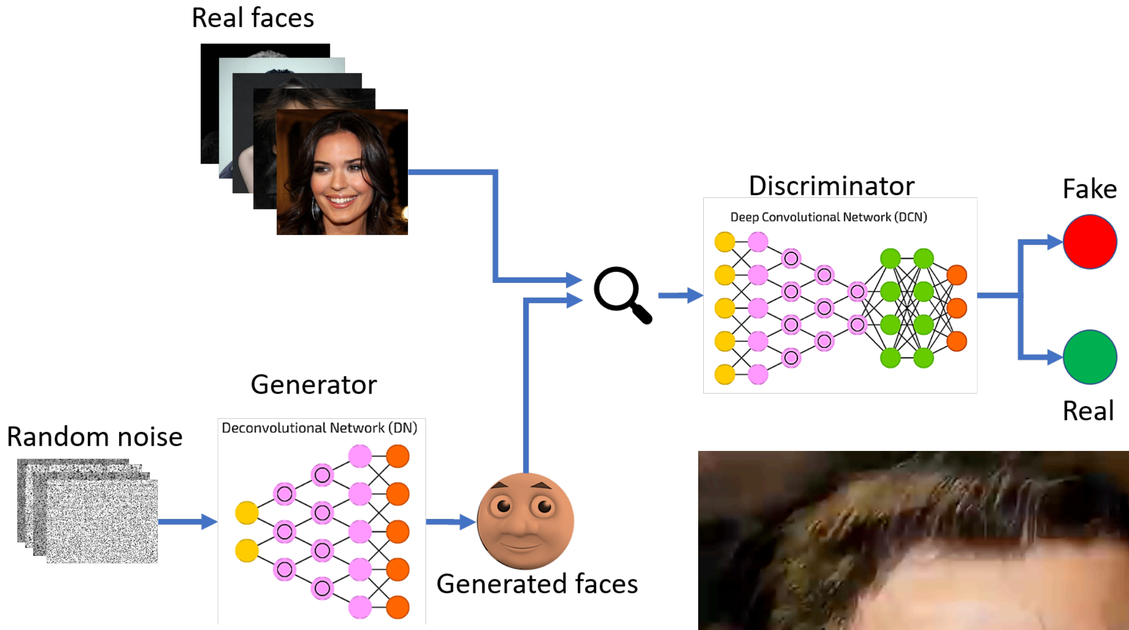
Yet Another Book

- *Today we enter newer territory and we depart from the DeepLearning book*
- *We will take this newer book as a reference*
  - *Full pdf available at the link, in case*



Generative Adversarial Networks

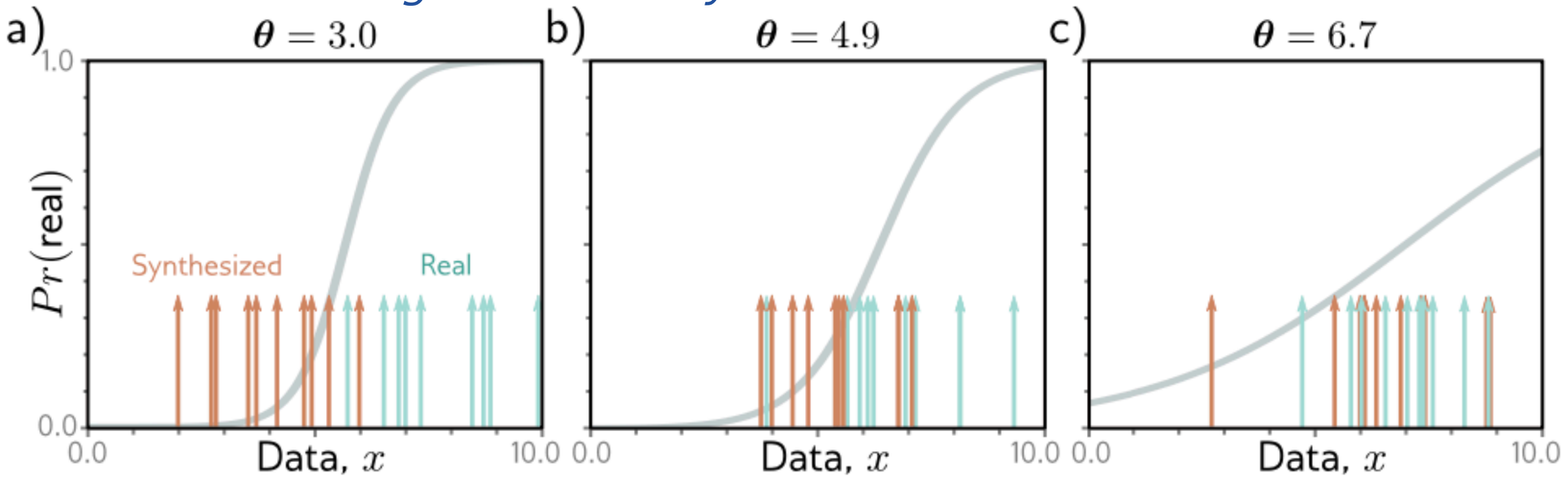
- *A GAN is a network that aims at creating new samples, undistinguishable from those contained in a given reference dataset*
- *GANs are trained through a two-network construction*
  - *A Generator: starts from random noise and generate the candidate new examples*
  - *A discriminator: receives an example (real or from generator, we will call it "fake") and has to identify which are the real ones*



https://thispersondoesnotexist.com/

Generative Adversarial Networks

- *The generator:*
  - *Starts from some latent space z*
  - *Creates the data x applying a network on z*
  - *The generator is a map from the space of z to that of x*
- *The discriminator:*
  - *A traditional classifier*
- *The GAN construction defines similarity between generated and real data through their statistical indistinguishability*



The GAN loss function

We start from the binary cross entropy of the discriminator

$$\mathscr{L} = \sum_i -(1-y_i)\log(1-\hat{y}_i) - y_i \log(\hat{y}_i)$$

where y_i is the ground-truth label and $\hat{y}_i = D(x_i|\phi)$, ϕ indicating the network parameters we want to determine

- We look for

$$\hat{\phi} = \text{argmin}_\phi \sum_i -\log(1-D(x^*|\phi)) - \log(D(x|\phi))$$

fixing $y_i = 1$ for real examples (labelled x), 0 for the fake ones (labelled as x^*)

- We now explicitly impose that x^* come from the generator $x^* = G(z|\theta)$ and we fix θ so that G maximizes confusion (i.e., increases the loss) of the discriminator

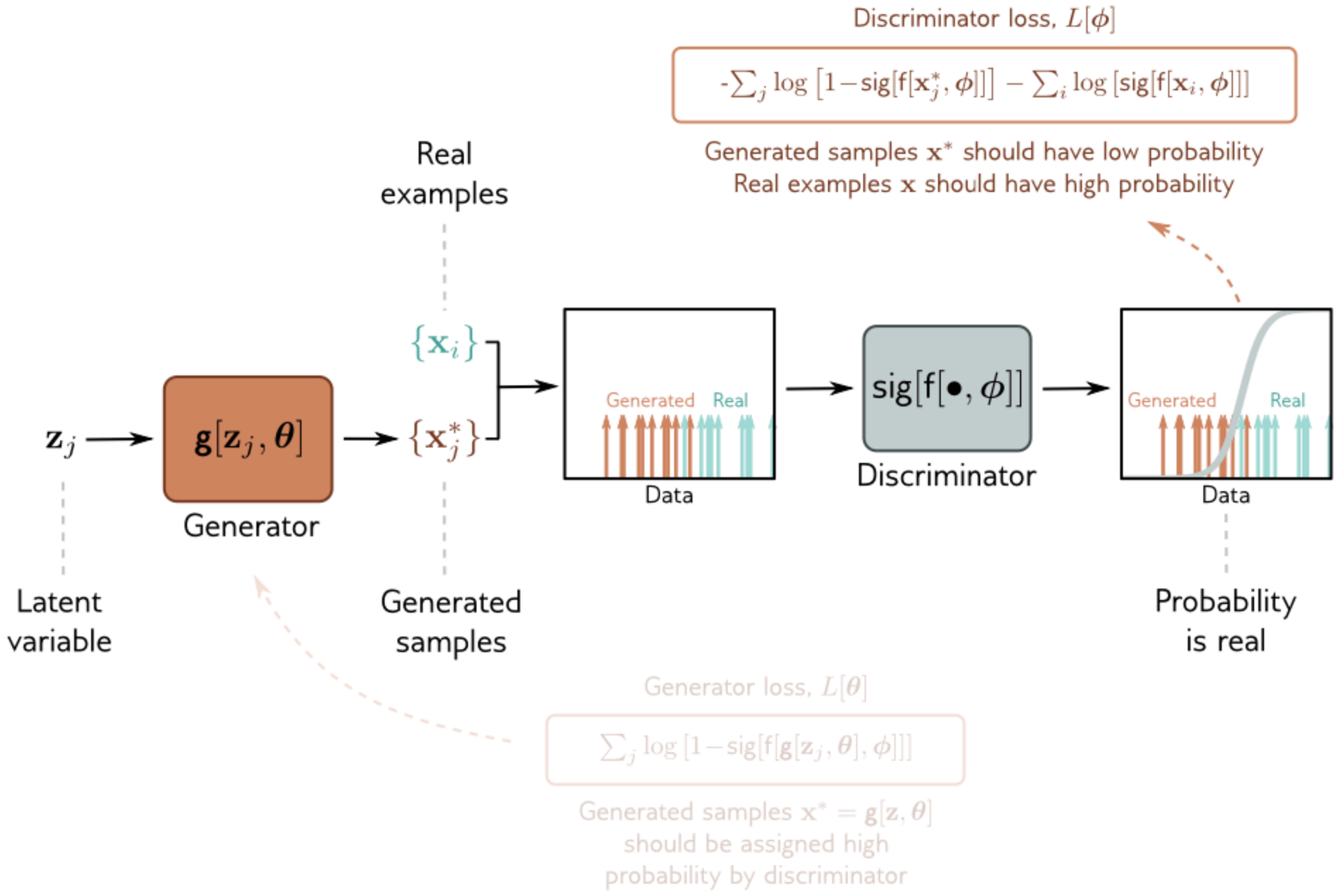
$$\hat{\phi} = \text{argmax}_\theta \left[\text{argmin}_\phi \sum_i -\log(1-D(G(z|\theta)|\phi)) - \log(D(x|\phi)) \right]$$

The GAN loss function

- More complicated than what we saw so far
  - MinMax game converging to a so-called Nash equilibrium: a solution which is the maximum of a function and the minimum of another one
  - Training is more delicate (and unstable)
  - It requires a specific procedure to handle this double task on two networks
- To define such a procedure, we define two loss functions to be minimized (notice the minus sign on $\mathscr{L}_\theta$)
  - $\mathscr{L}_\theta = \sum_i \log(1 - D(G(z|\theta)|\phi))$ **Trains the Generator**
  - $\mathscr{L}_\phi = \sum_i -\log(1 - D(G(z|\theta)|\phi)) - \log(D(x|\phi))$ **Trains the Discriminator**

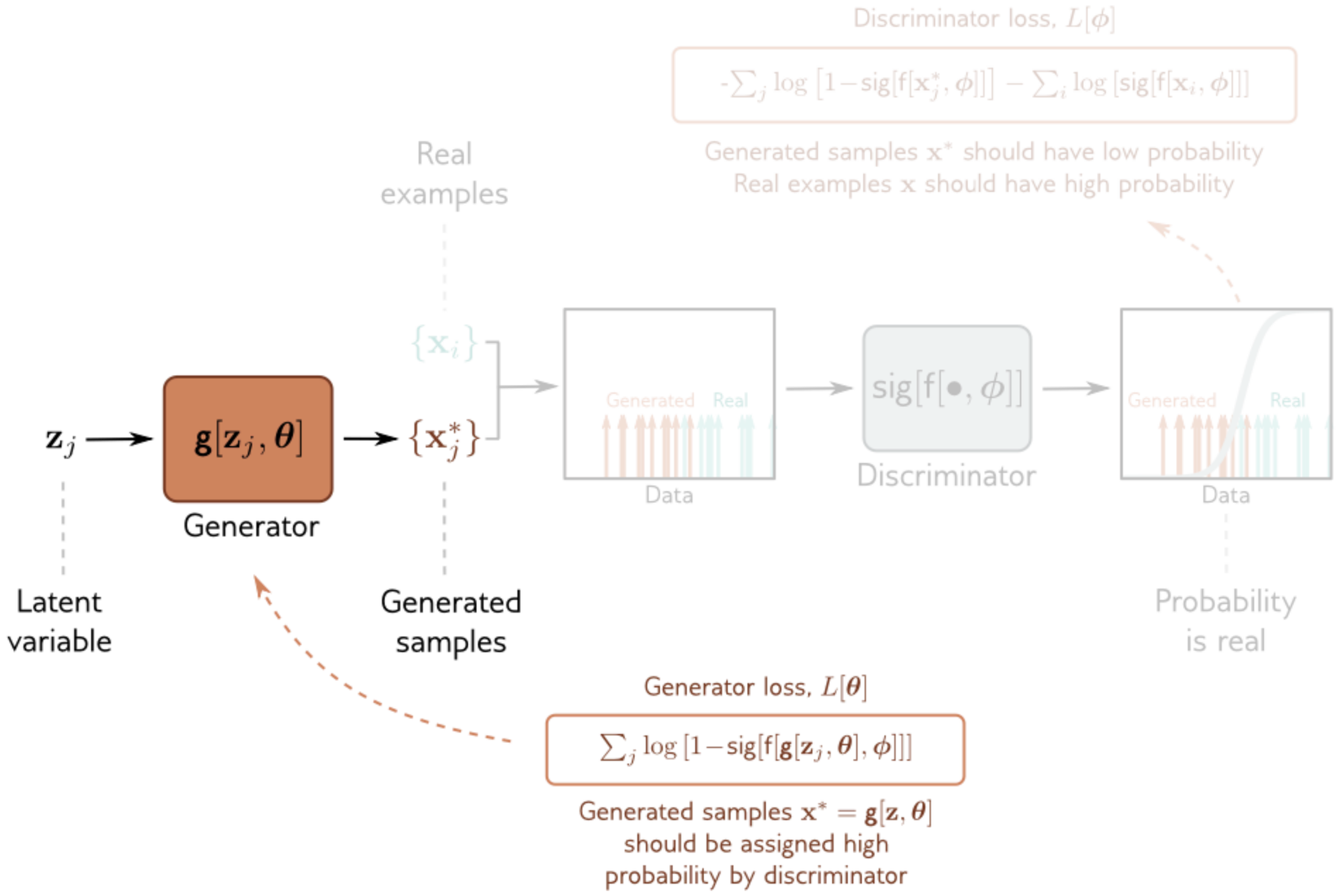
Training a GAN

- *Step 1: for a given batch*
  - *Release the Discriminator parameters*
  - *Freeze the Generator parameters*
  - *Train the Discriminator minimizing* $\mathscr{L}_\phi$



Training a GAN

- *Step 2: for a given batch*
  - *Freeze the Discriminator parameters*
  - *Release the Discriminator parameters*
  - *Train the Generator minimizing* $\mathscr{L}_\theta$



Training a GAN

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

end for

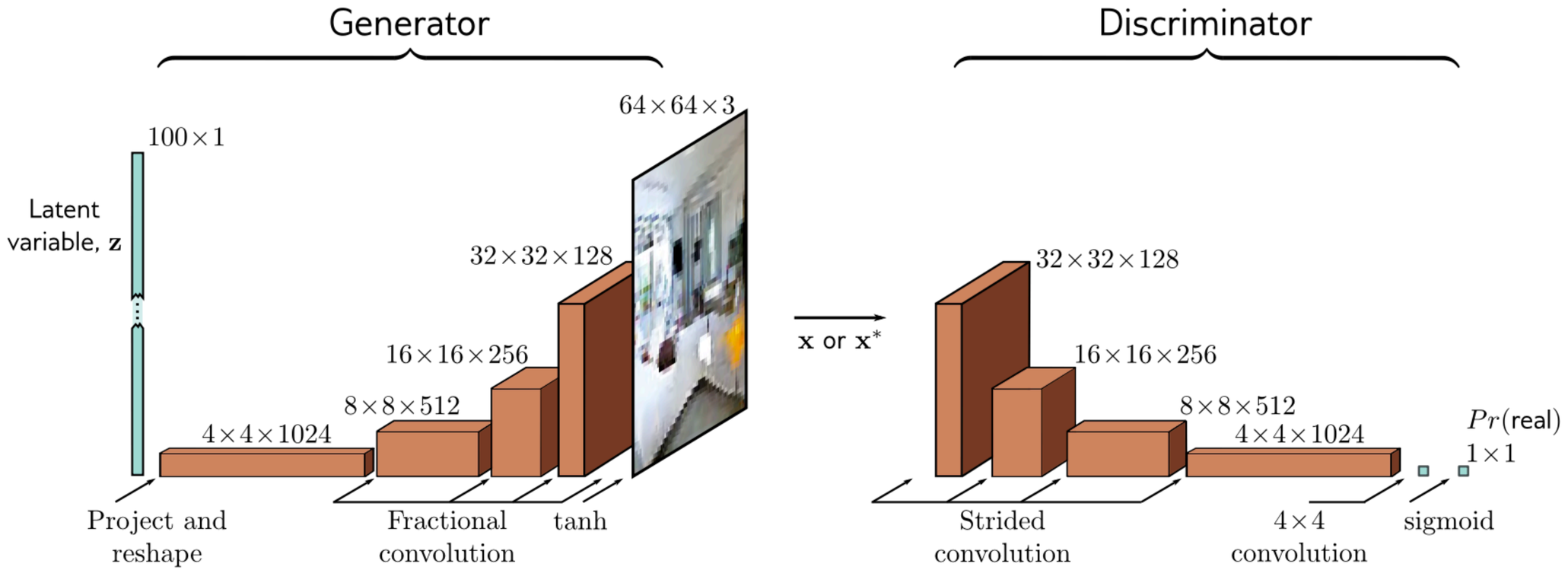
- Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

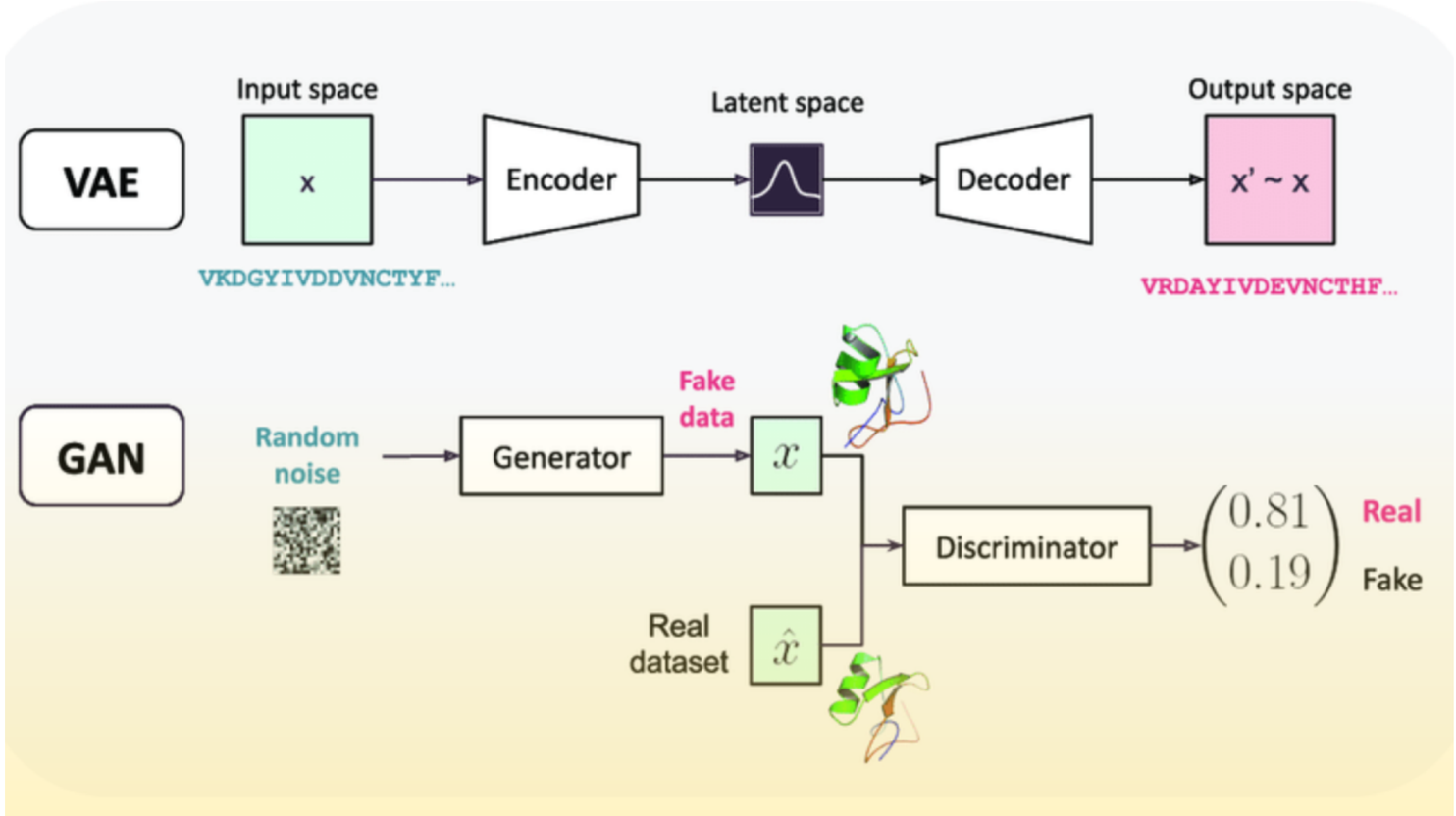
Example: DC Gan

- *A Convolutional Generator from random noise to images*
- *A convolutional binary classifier from images to probability(true)*



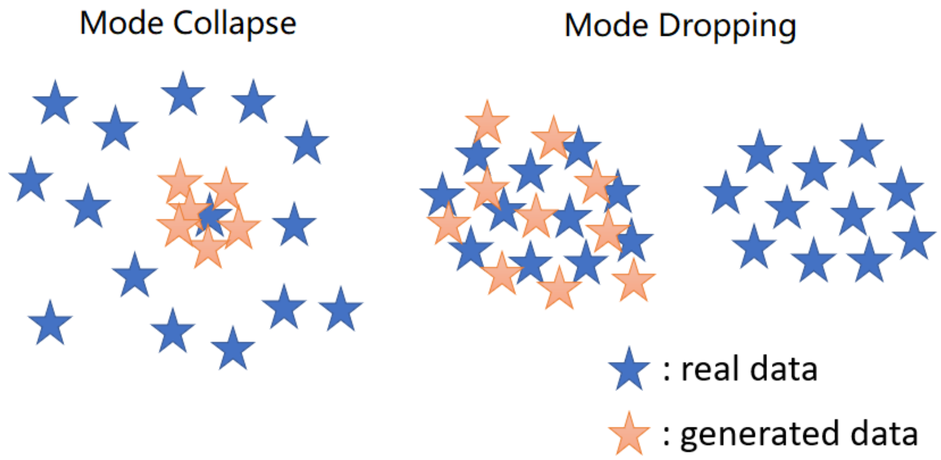
Differences Between GANs and VAEs

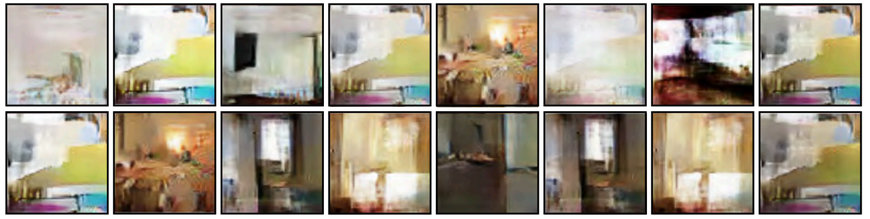
- *VAEs use data to determine the pdf of the latent space variables*
  - *New examples can be sampled from it*
- *GANs learn a map from latent space to data space*
  - *examples can be generated, but not sampled (we don't have the measure of how probable each point in the latent space is)*

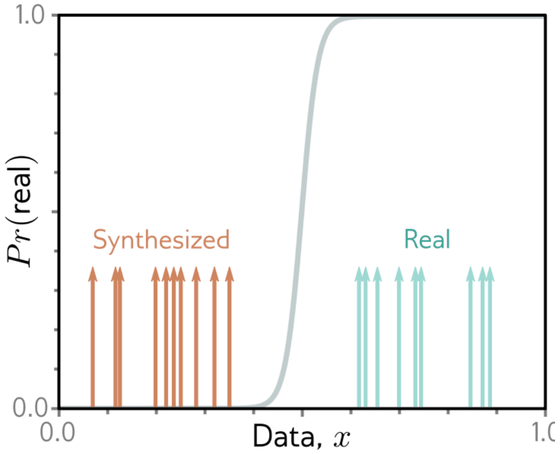


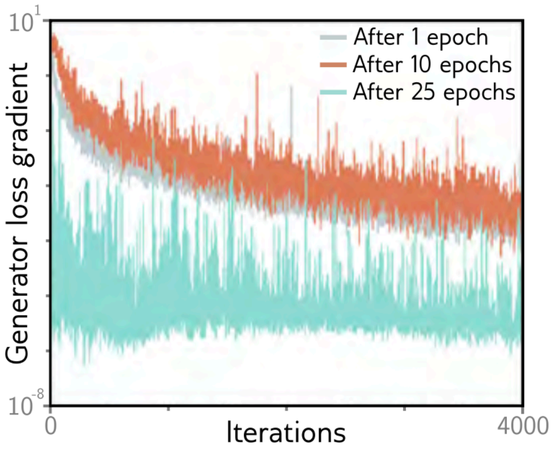
Training a GAN is practice

- A GAN training can fail in many ways
  - Mode dropping: converge to plausible generations, but only spanning a subset of the data types (e.g., generate faces, but never with specific features, like beards)
  - Mode collapse: the network ignores the randomness of the input and generates always the same image with minimal variation
  - Vanishing gradient: if (particularly at beginning) the generator gives easy-to-identify fake images, a small change in the generator will not overcome the easy task of the discriminator. Happens whenever the pdf (in z) of the real and generated images is very different (very often).
- To avoid these issues, people have to go to tricky trial&error procedures, resulting in often ad-hoc prescriptions
  - e.g., DCGAN required BatchNorm in G and D, stride>1, leaky ReLu in D, Adam optimizer with reduced momentum
  - A plan network would maybe not train optimal for different choices, but the training would not fail completely
- In general, one needs a good discriminator to challenge the generator, but not a too good discriminator









Wasserstein GANs

- A way to fix the GAN training instability is to change a different loss.

- Stable results were obtained very early, when moving to a different loss

$$\mathcal{L}[\phi] = \sum_j f(x_j^* \,|\, \phi) - \sum_i f(x_i \,|\, \phi) = \sum_i f(G(x_i \,|\, \theta) \,|\, \phi) - \sum_i f(x_i \,|\, \phi)$$

where the discriminator $f(x)$ is forced to have a gradient norm

$$\left| \frac{\partial f(x \,|\, \phi)}{\partial x} \right| < 1$$

- Let's see where this comes from

Wasserstein distance

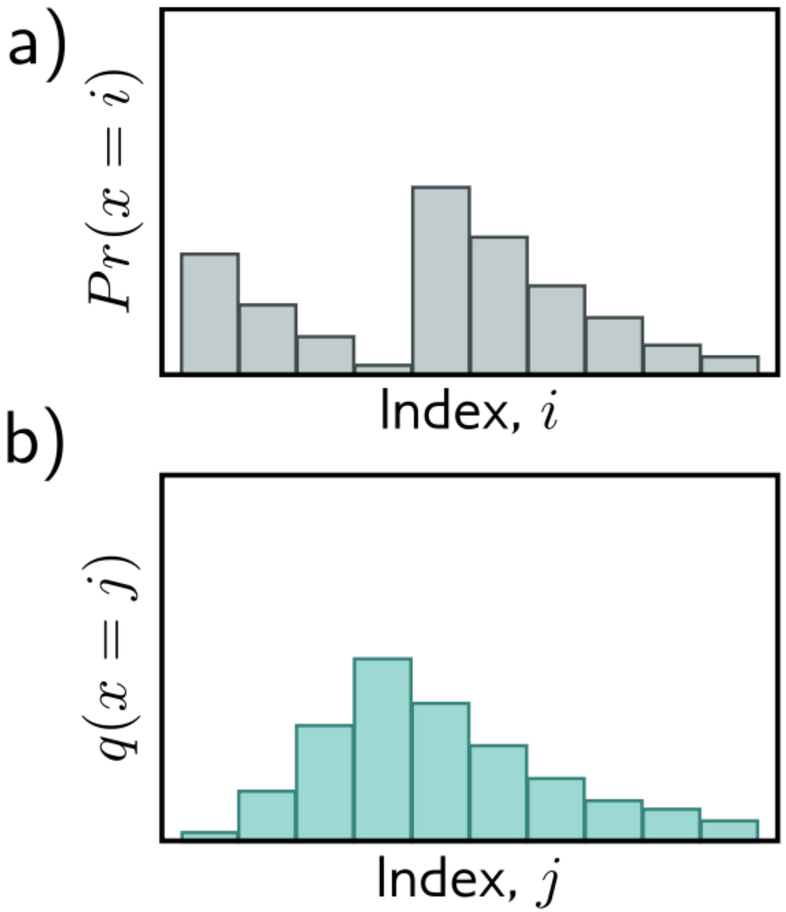
- The Wasserstein distance (or earth mover distance) is a measure of distance between functions
  - An alternative to other metrics, like the KL divergence (and JS distance) that we already discussed
- It corresponds to the work needed to move probability mass from a function f to make it coincide with some other function g
- For discrete quantities, it is defined as

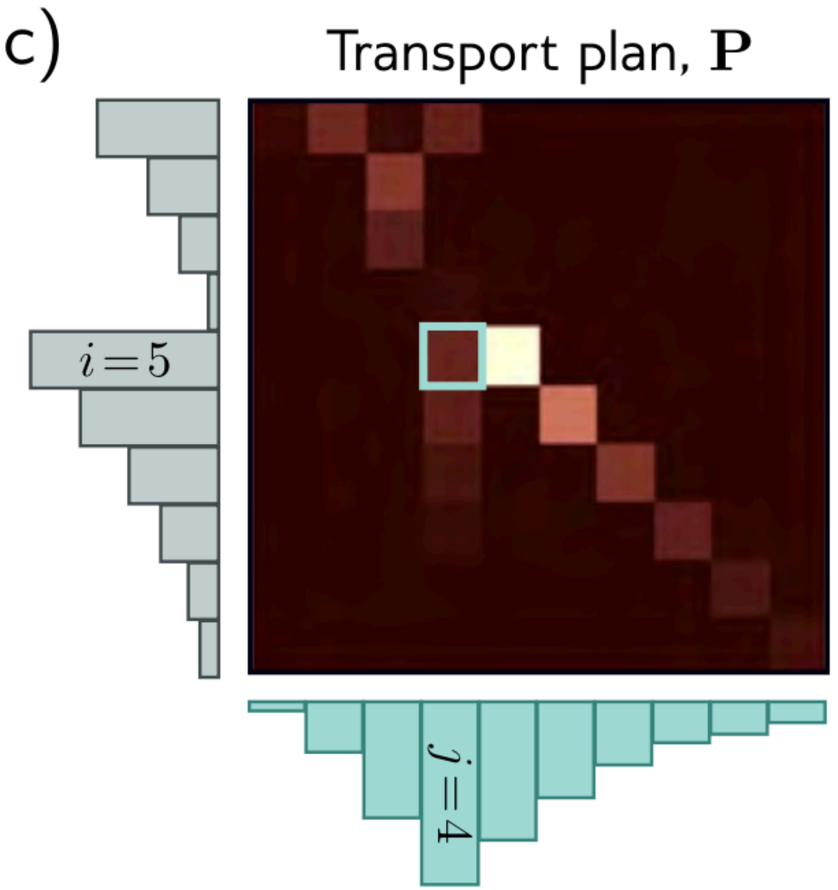
$$D_w[Pr_i, q_j] = \min_P \left[\sum_{i,j} P_{ij} |i - j| \right]$$

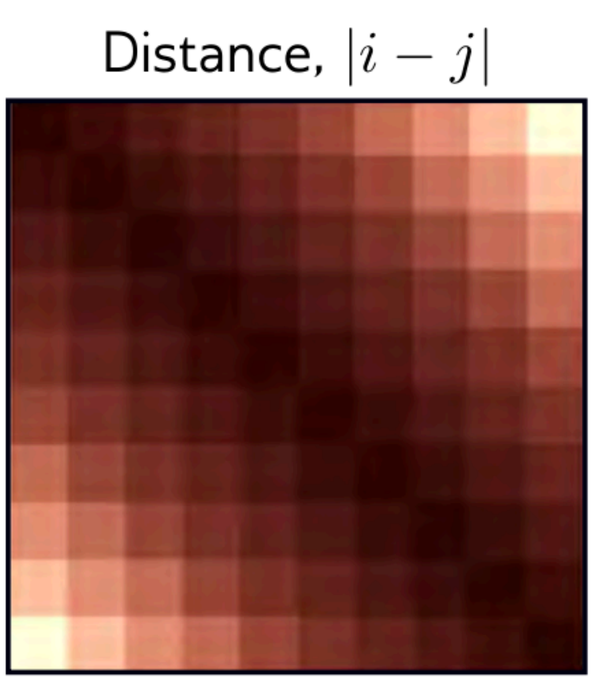
where the matrix P store the amount of probability moved from q to Pr, defined such that

$$\sum_j P_{ij} = Pr(i) \ , \ \sum_i P_{ij} = q_j \ , \ P_{ij} \geq 0$$

Wasserstein distance







Wasserstein distance
$= \sum \mathbf{P} \cdot |i - j|$

Wasserstein distance

- It can be shown (we skip the math there) that given a set of discrete functions f such that

$$D_w[Pr, q] = \max_f \left[\sum_i Pr_i f_i - \sum_j q_j f_j \right]$$

with $|f_{i+1} - f_i| < 1$

- This can be generalized to continuous functions as

$$D_w[Pr(x), q(x)] = \min_\pi \left[\int\int \pi(x_1, x_2) \, |x_2 - x_1| \, dx_1 dx_2 \right]$$

which can computed maximizing the following expression over an ensemble of functions

$$D_w[Pr(x), q(x)] = \max_{f(x)} \left[\int Pr(x) f(x) dx - \int q(x) f(x) dx \right]$$

with the same bound on the gradient

Wasserstein Loss

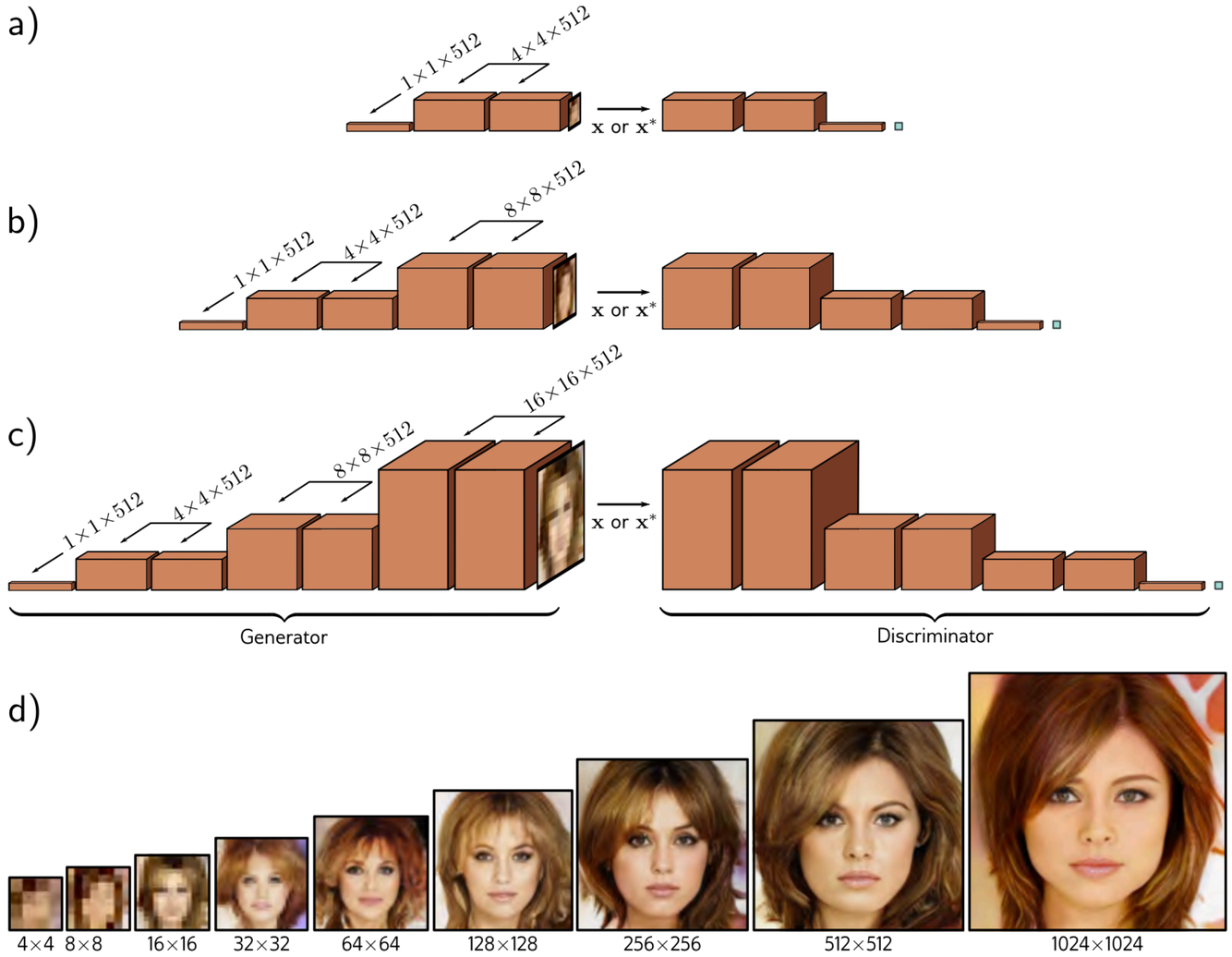
- Our loss is then understood in terms of the Wasserstein distance
  - We optimize over the ensemble of functions f that can be expressed by our neural network f
  - Pr is the pdf of the generated samples
  - q is the pdf of the real samples
  - We approximate the two integrals as numerical sums over the examples in the generated and real dataset
- We then obtain

$$\mathscr{L}[\phi] = \sum_j f(x_j^* \,|\, \phi) - \sum_i f(x_i \,|\, \phi) = \sum_i f(G(x_i \,|\, \theta) \,|\, \phi) - \sum_i f(x_i \,|\, \phi)$$

and the gradient penalty comes from he gradient bound of the Wasserstein distance

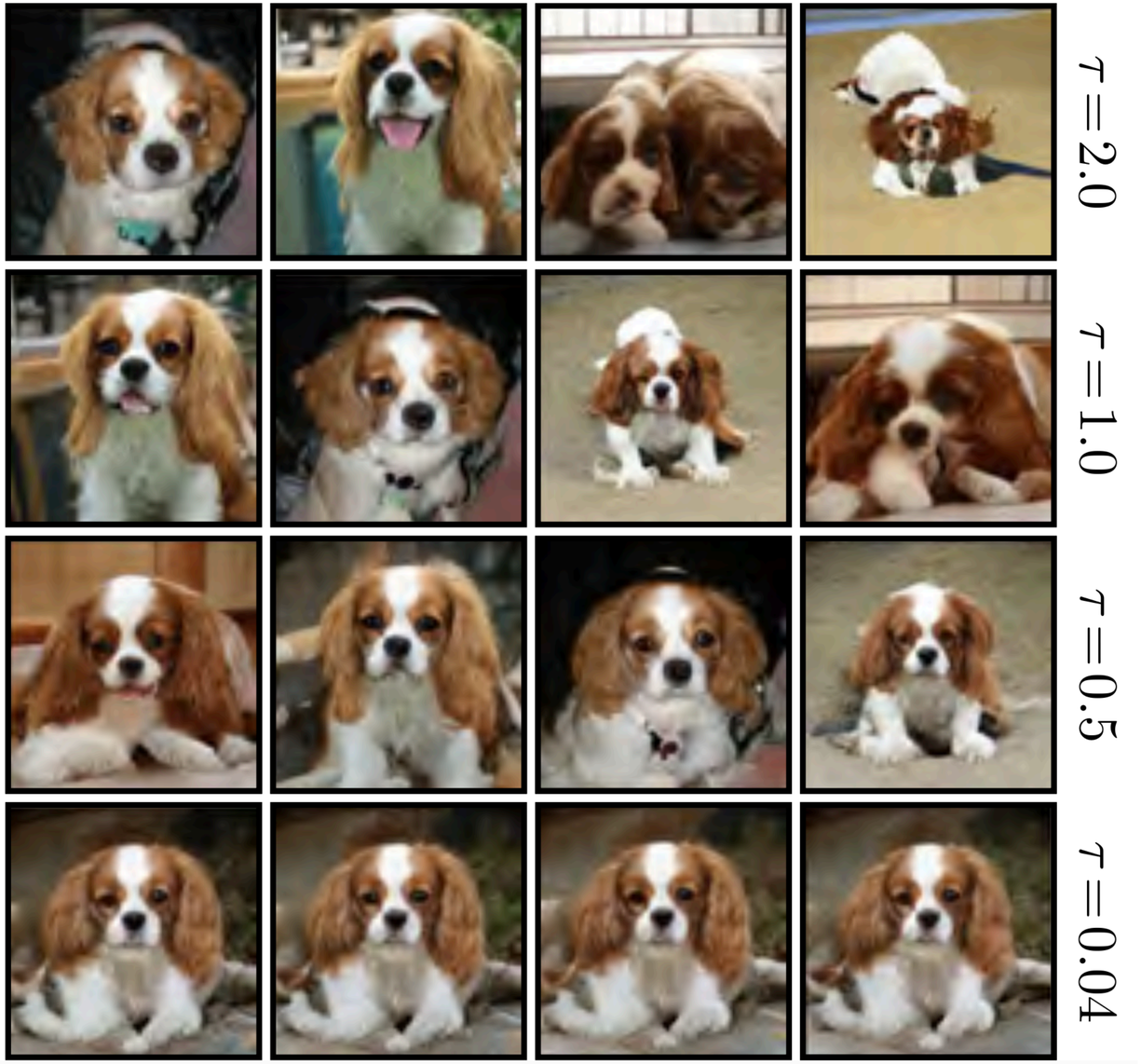
Training strategies

- *Progressive growing: one starts with low resolution (4x4) images, and progressively moves to higher resolution adding upsampling layers to the generator and extra "first" layers to the discriminators*
  - *This approach opens the way to high-resolution generators*



Training strategies

- *Truncation: z are sampled around their high probability (approximated by the mean). Less variation in the sample, but higher resolution is obtained*

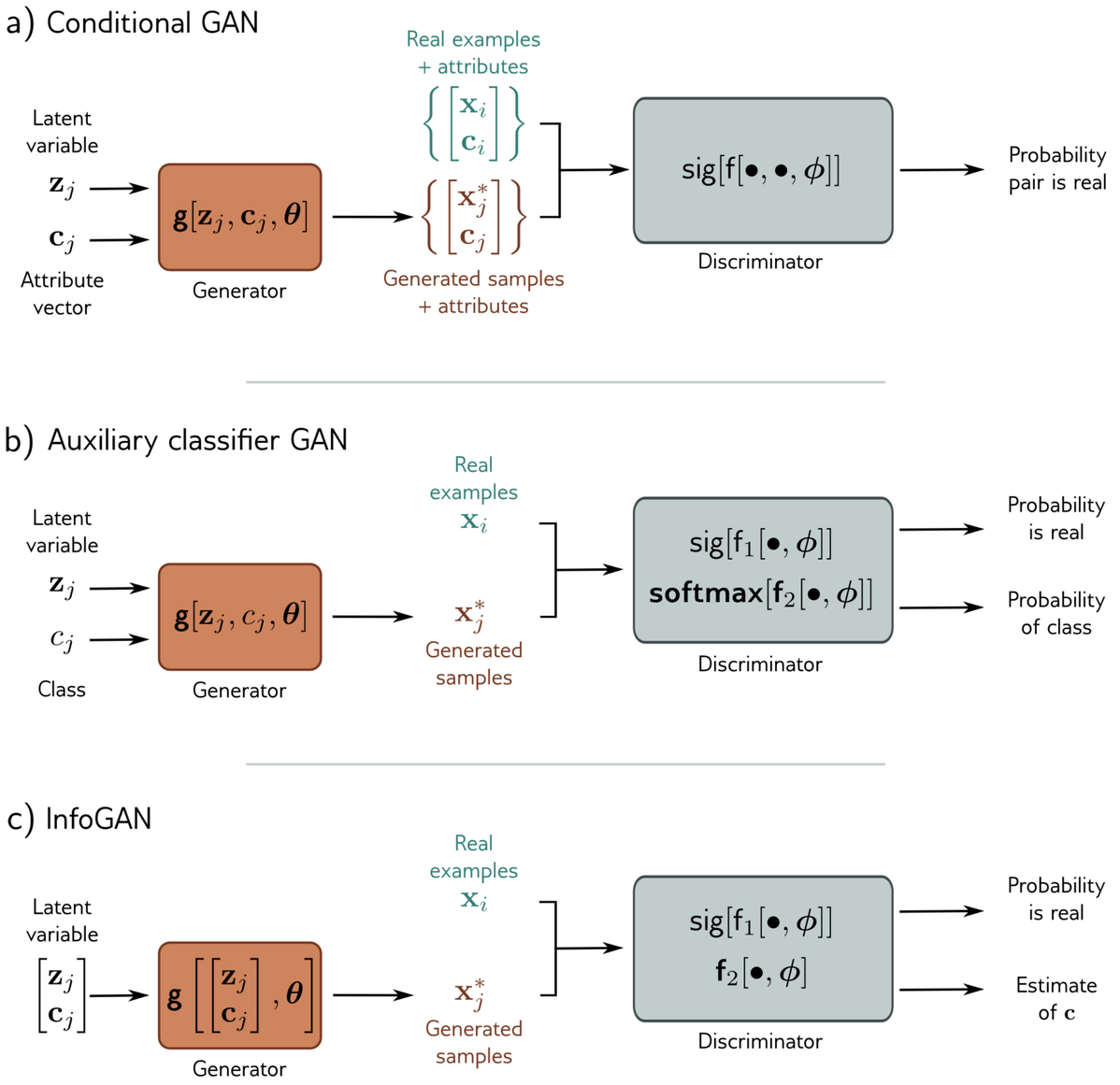


Training strategies

- *Minibatch discrimination: the discriminator is provided with additional information, corresponding to the variation of the input across a mini batch*
  - *e.g., the pixel-by-pixel L1 norm between inputs*
- *With that, the discriminator can easily identify generated images that are too similar to each other (small variation) and force the generator to diversify the images*
  - *This pushes the generator far from mode collapsing*

Conditional Generation

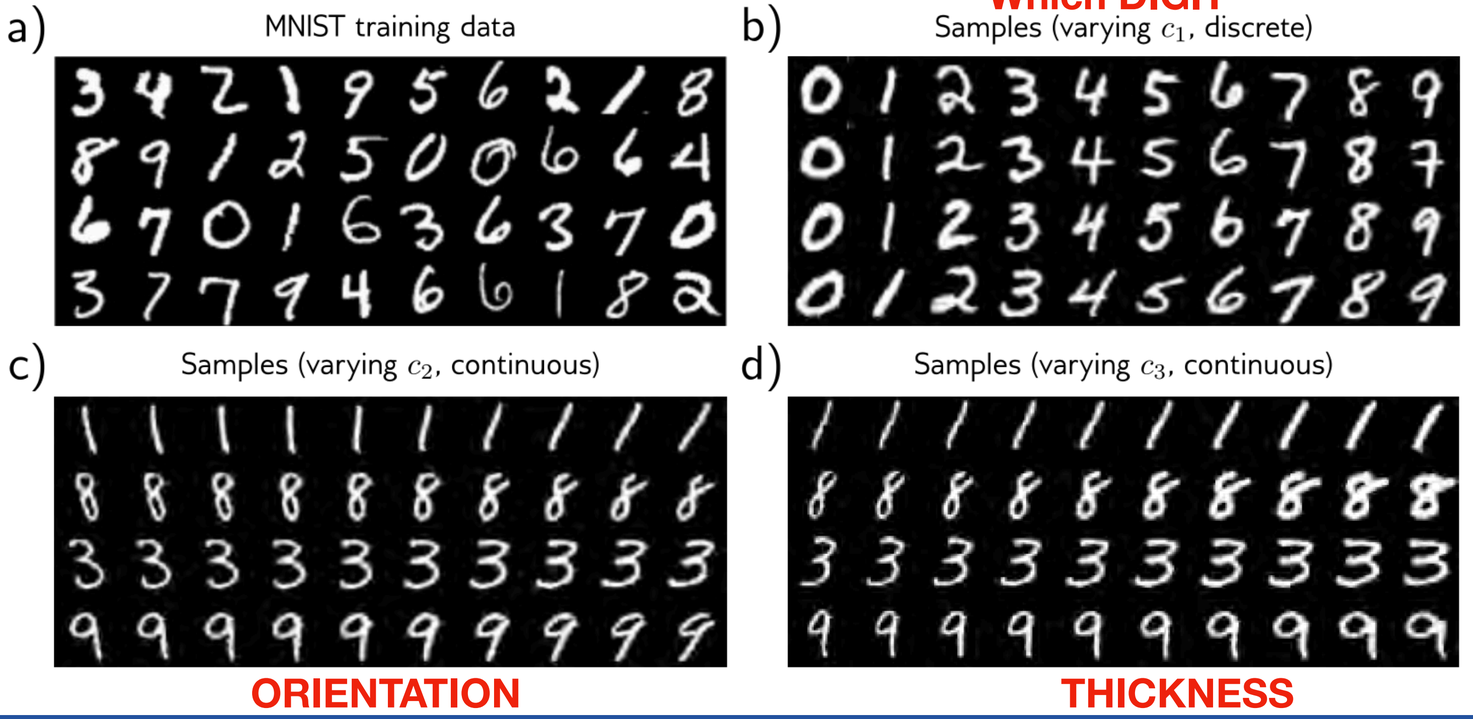
- *One can force the GAN towards specific tasks, injecting information as part of the latent-space representation*
  - *Conditional GANs: additional attributed to Generators, passed fwd to the discriminator. Forces generating specific aspects*
  - *Auxiliary GANs: similar to conditional GANs, but for discrete classes. Class is not passed to discriminator, who has to guess it (double classification task)*
  - *InfoGAN: formally similar to Auxiliary GAN, but the input information is random: in an unsupervised fashion, the network learns to associate it to specific features*



Conditional Generation

- *Example of Auxiliary GAN for ImageNet: can be used to ask not a generic image, but of a specific class (e.g., "generate an image of a monarch butterfly")*
- *Example of InfoGAN: additional info can be discrete or continuous*

Which DIGIT

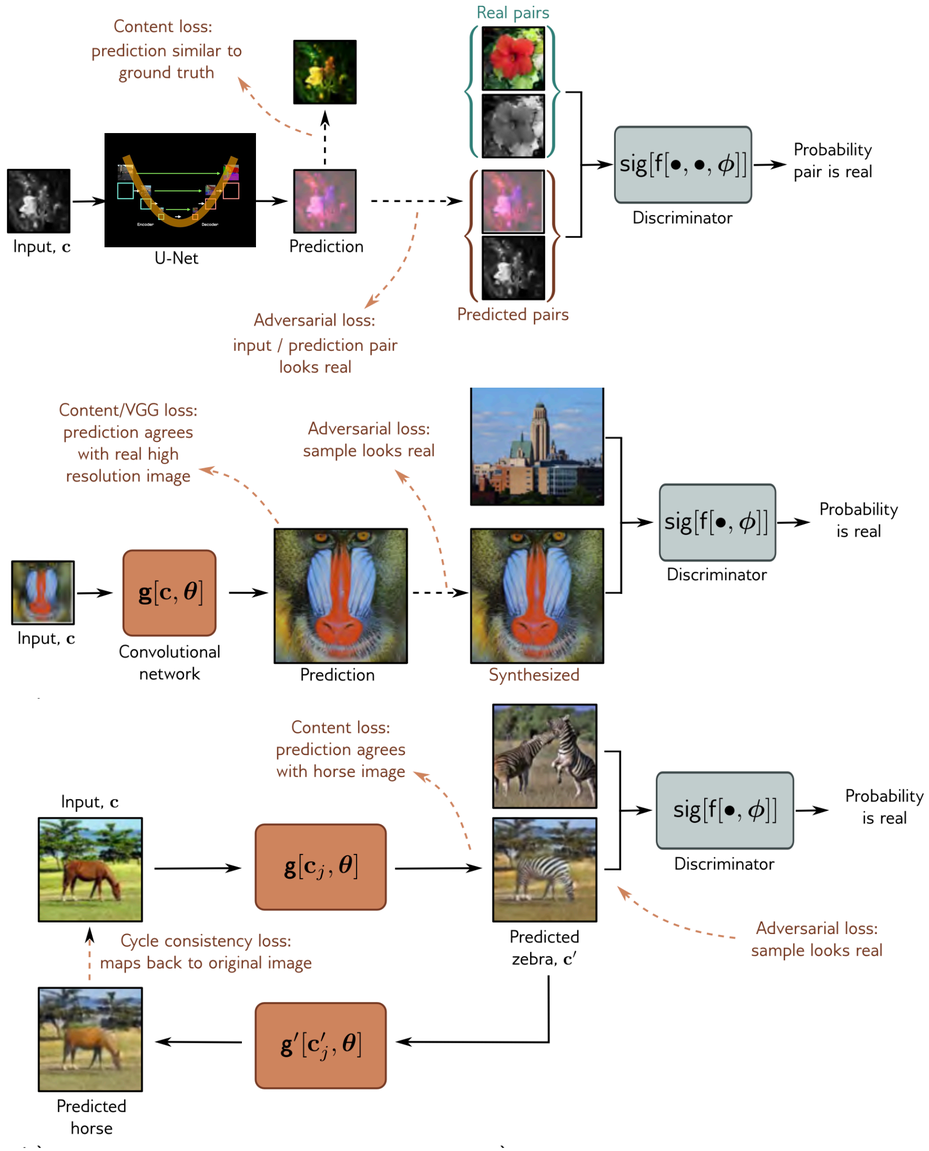


ORIENTATION

THICKNESS

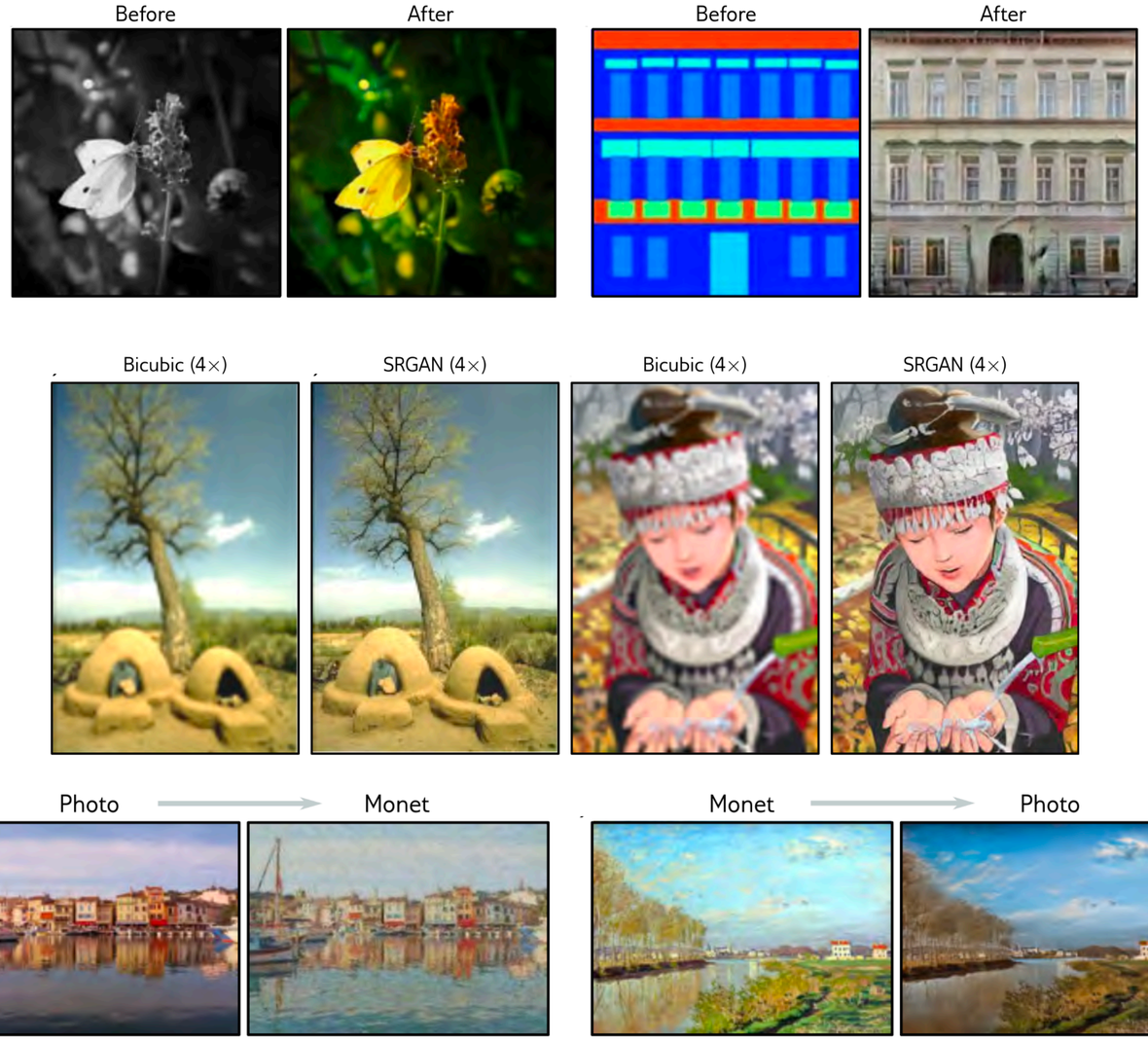
Other Usage of Adversarial Training

- *Pix2Pix is a U-Net that applies style to images (e.g., adding colors to b&w images)*
- *SRGAN returns high-resolution images from low-resolution ones*
- *CycleGAN applies style from reference dataset to a given dataset. It is trained without having examples of before/after correspondence (Monet pairings, images, but no images of landscapes painted by Monet)*



Other Usage of Adversarial Training

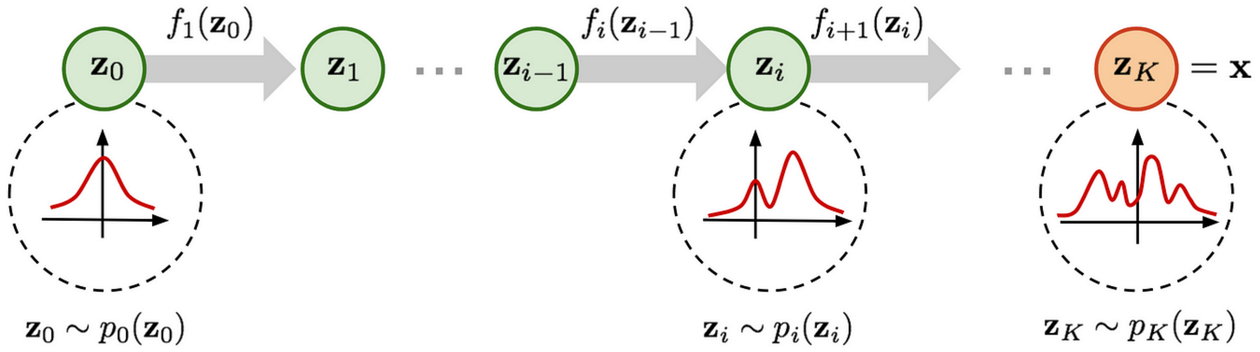
- *Pix2Pix is a U-Net that applies style to images (e.g., adding colors to b&w images)*
- *SRGAN returns high-resolution images from low-resolution ones*
- *CycleGAN applies style from reference dataset to a given dataset. It is trained without having examples of before/after correspondence (Monet pairings, images, but no images of landscapes painted by Monet)*



Beware: GANs hallucinate

Going Beyond GANs

- 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

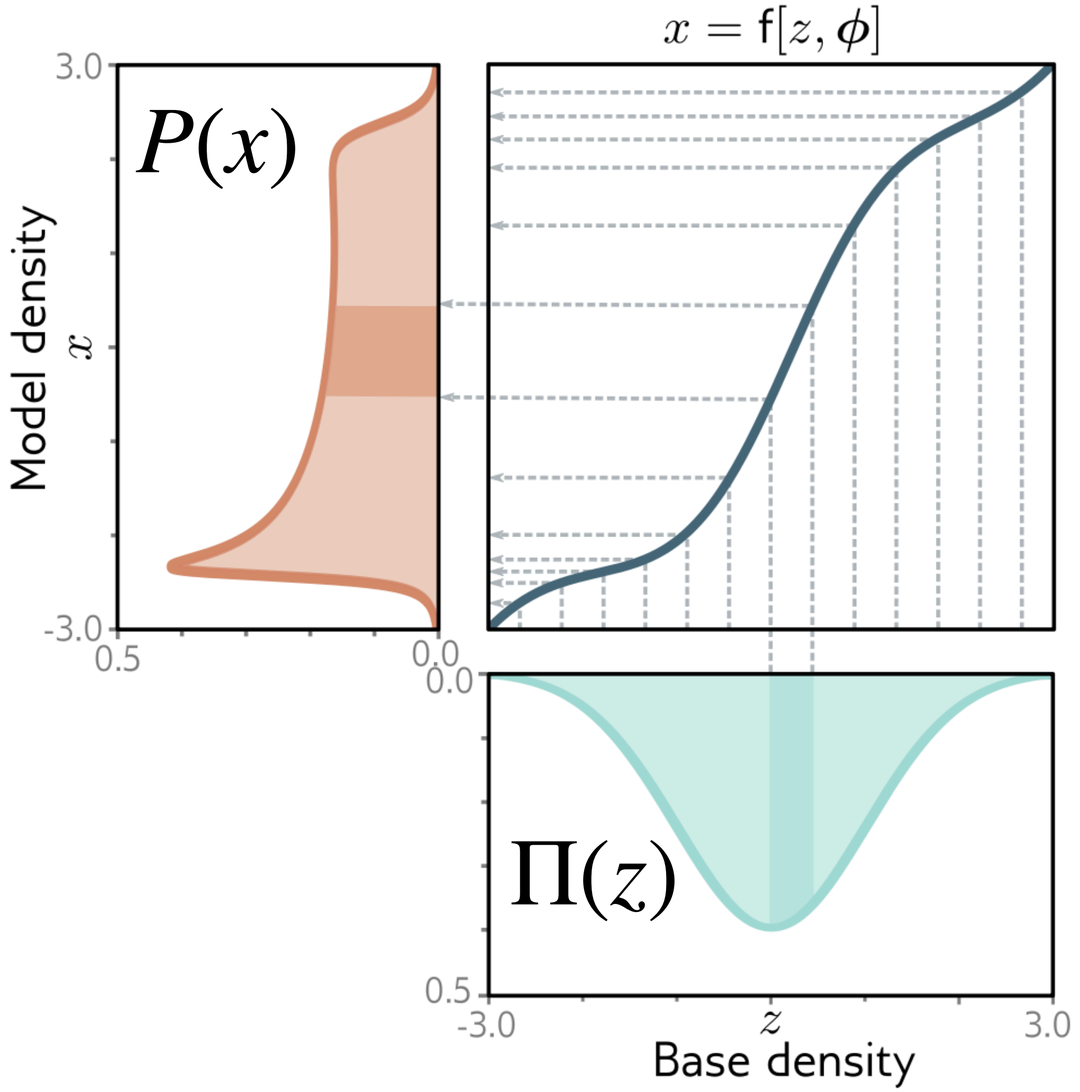
$$\int \Pi(z)dz = \int \Pi(z)\left|\frac{dz}{dx}\right| dx = \int \Pi(f^{-1}(x))\left|\frac{df^{-1}(x)}{dx}\right| dx = \int P(x)dx$$

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



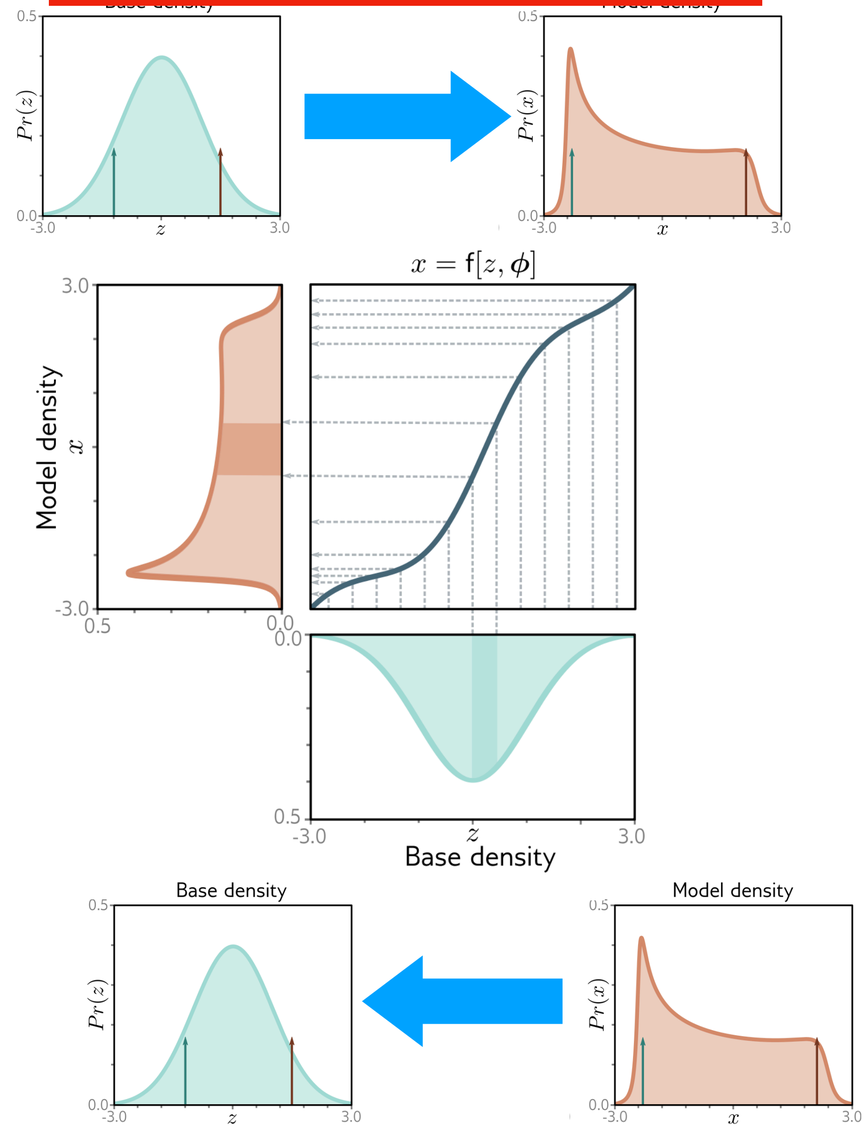
1D Normalizing Flow

- 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

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

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

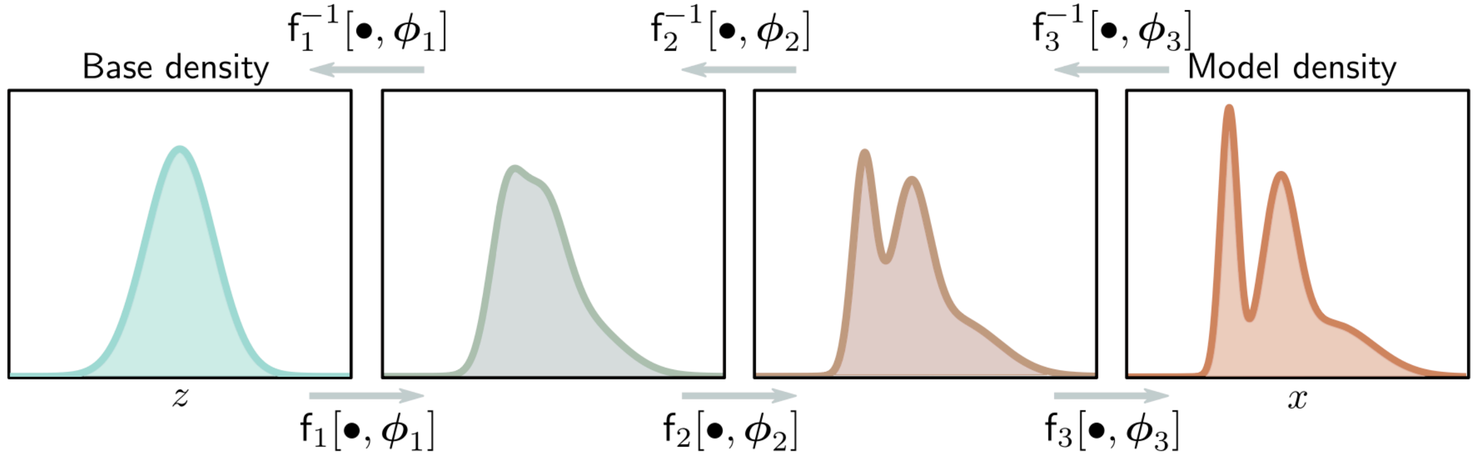
$$\text{argmin}_\phi \left[\sum_i \log \left| \frac{\partial f(z_i, \phi)}{\partial z_i} \right| - \sum_i \log \Pi(z_i) \right]$$

- 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 $P(x)$*

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



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

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

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
  - 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*

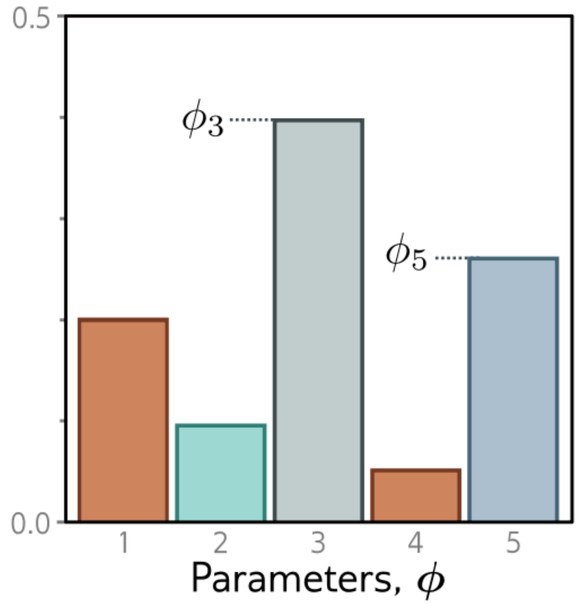
$$\mathbf{f}[\mathbf{h}] = \Big[\mathrm{f}[h_1, \boldsymbol{\phi}], \mathrm{f}[h_2, \boldsymbol{\phi}], \ldots \mathrm{f}[h_D, \boldsymbol{\phi}]\Big]^T$$

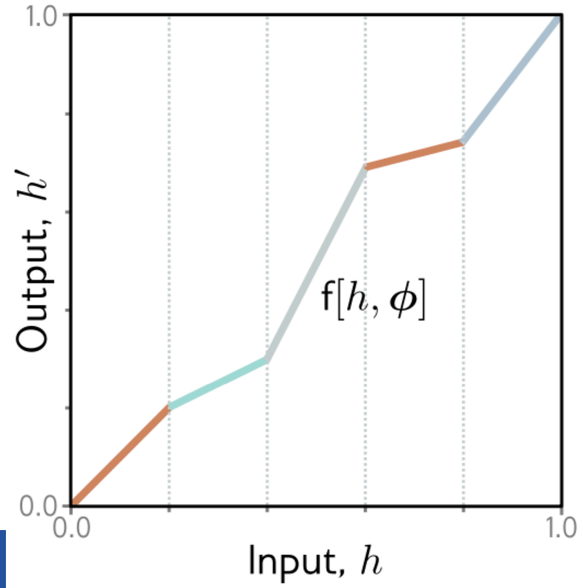
- *The jakobian is diagonal, since there in no cross-talk between the elements in the transformation. So the determinant calculation is easy*

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

- *It adds non-linearity, but it cannot alter correlations between variables*
- *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)*

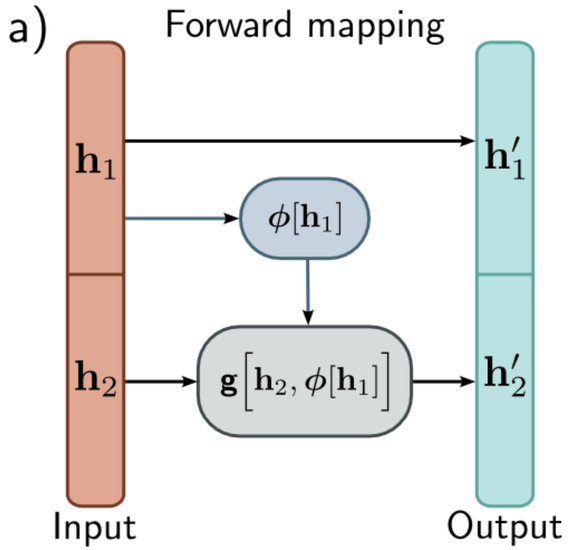
$$\mathrm{f}[h, \boldsymbol{\phi}] = \left(\sum_{k=1}^{b-1} \phi_k\right) + (hK - b)\phi_b$$



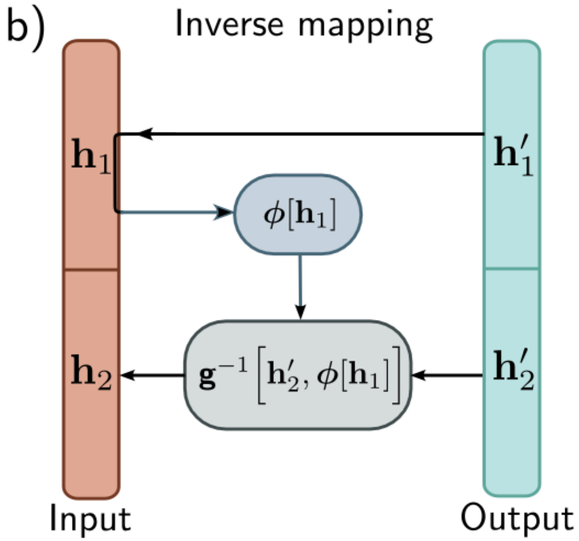


Coupling flows

- Couplings flows operate as follows
  - Split input h in two vectors h_1 and h_2
    - h_1 is passed as it is to output
    - 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
  - 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)



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



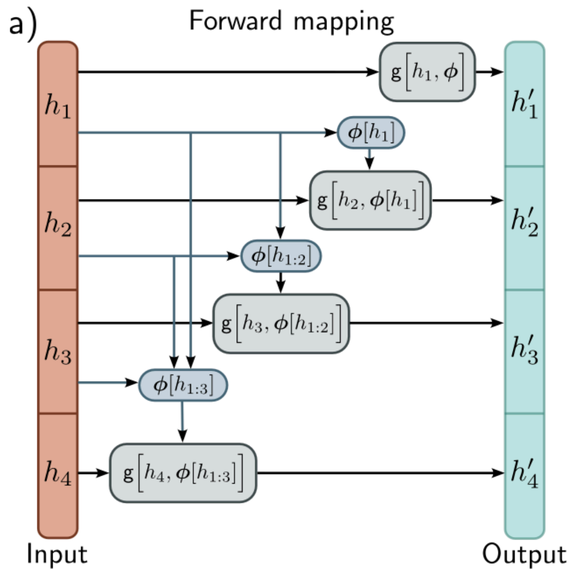
$$\mathbf{h}_1 = \mathbf{h}'_1$$
$$\mathbf{h}_2 = \mathbf{g}^{-1}\left[\mathbf{h}'_2, \phi[\mathbf{h}_1]\right]$$

Autoregressive flows

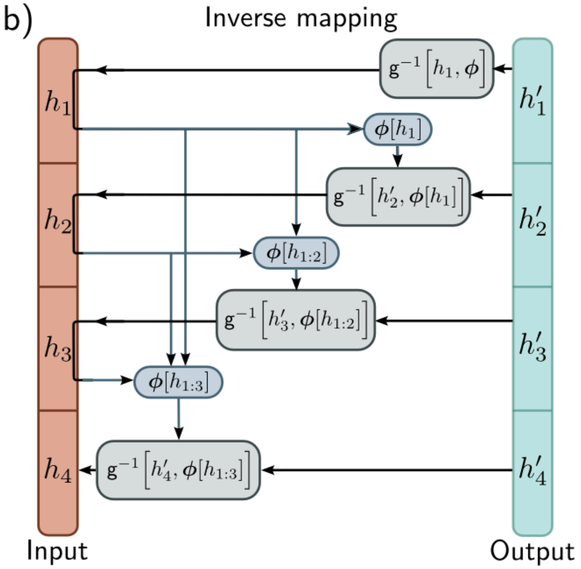
- *Generalize coupling flows to enhance complexity and expressivity*
  - *Each input is a separate block*
  - *Each block is transformed according to a transformer function $g[h_i, \phi]$ (not related to transformer networks!!!), depending on conditioners $\phi = [\phi_{1:i}]$*

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

- *Notice: since the computation depends 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*



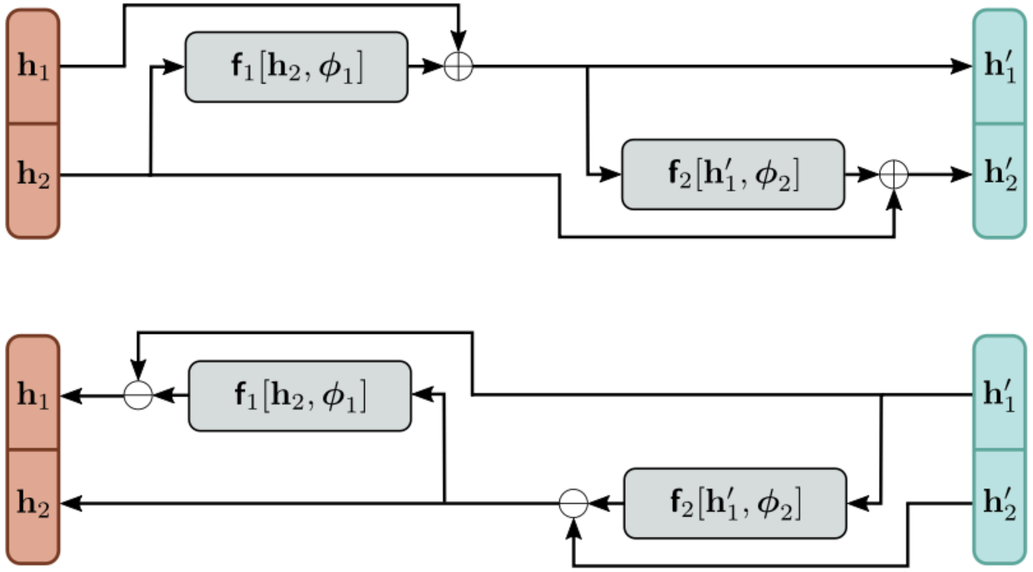
$$\begin{aligned} h'_1 &= \mathrm{g}\Big[h_1, \boldsymbol{\phi}\Big] \\ h'_2 &= \mathrm{g}\Big[h_2, \boldsymbol{\phi}[h_1]\Big] \\ h'_3 &= \mathrm{g}\Big[h_3, \boldsymbol{\phi}[h_{1:2}]\Big] \\ h'_4 &= \mathrm{g}\Big[h_4, \boldsymbol{\phi}[h_{1:3}]\Big] \end{aligned}$$



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

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* $h_2' = g[h_1, \phi_2] + h_2$

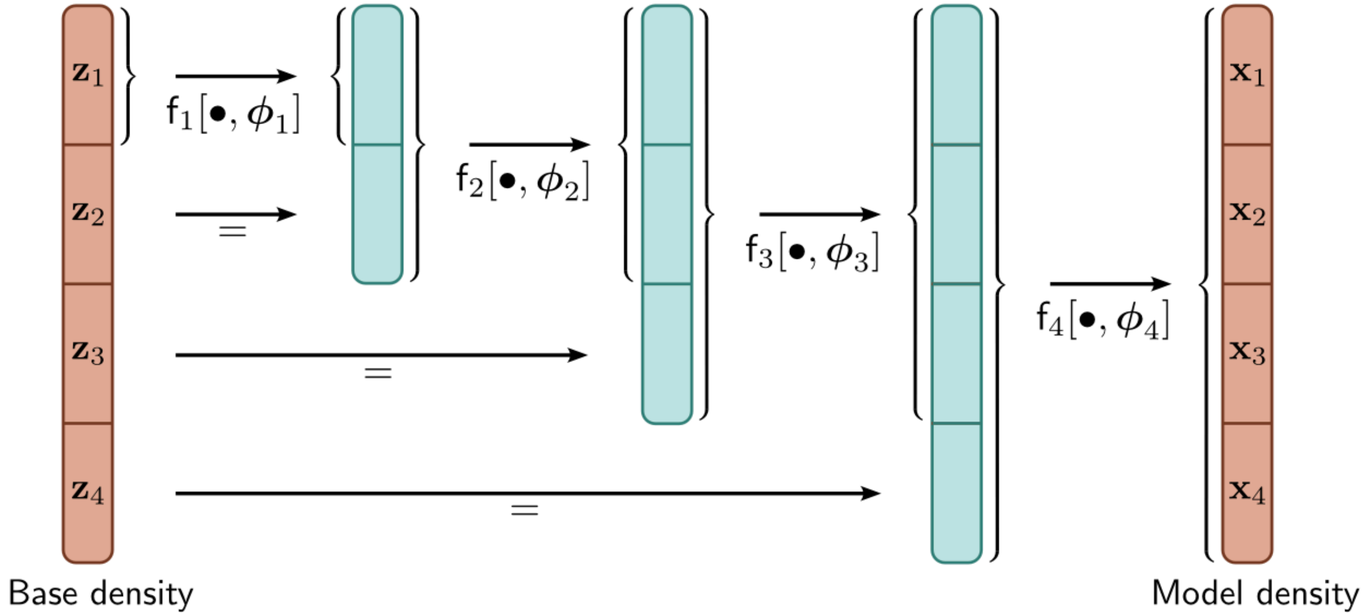


$$\begin{aligned} \mathbf{h}_1' &= \mathbf{h}_1 + \mathbf{f}_1[\mathbf{h}_2, \phi_1] \\ \mathbf{h}_2' &= \mathbf{h}_2 + \mathbf{f}_2[\mathbf{h}_1', \phi_2]; \end{aligned}$$

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

Multi-scale flows

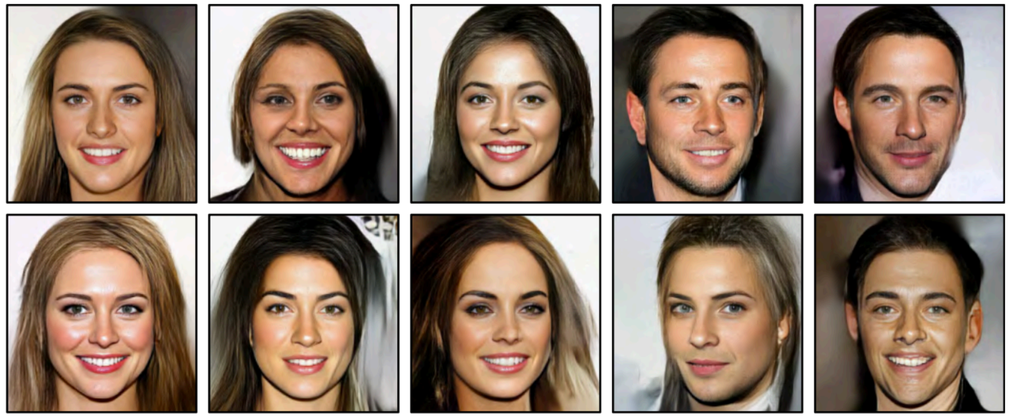
- *To be invertible, flows need to have outputs of the same dimension as inputs*
- *On the other hand, some part of the computation might only depend on a subset (in which case, carrying around the entire input when only a faction is needed would be inefficient*
- *In multi-scale flows, this is achieved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