Modelling High-dimensional Data: Manifolds, Autoencoders, and Generative Adversarial Networks

Volker Tresp

Winter 2024-2025

Part I: Mathematical Introduction

From Conditional Distribution to Unconditional Distributions

- In classification and regression we were modelling

$$P(y|x_1, \ldots, x_M)$$

- How about modelling

$$P(x_1, \ldots, x_M)$$

  (you can consider the y-variable as one of the inputs)

- In a way this model is more powerful, since using marginalization and conditioning, we can derive any

$$P(x_i|\{S_i\})$$

  where $\{S_i\} \subseteq \{x_1, \ldots, x_M\} \setminus x_i$

- Examples:
- x_i reflects the rating of users for movie i (recommendation engines)
- x_i is pixel i in an image

Bayes Nets

- Recall the chain rule

  $P(x_1, \ldots, x_M) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2) \ldots P(x_M|x_1, \ldots, x_{M-1})$

- Bayes nets use the decomposition

  $$P(x_1, \ldots, x_M) = \prod_{i=1}^{M} P(x_i|par(x_i))$$

  where $par(x_i)$ is a subset of the predecessors

- Interpretable (maybe even causally): a acyclic directed graph (DAG)
- Inference is based on marginalization and conditioning
- Works best if dependencies are actually low-dimensional (and can be determined and quantified by human experts)
- Powerful in modelling medical diagnostics

Markov Nets

- Consider a basis-function model

$$f(x_1, \ldots, x_M) = \sum_i w_i \phi_i(\{S_i\})$$

  where each basis function might only depend on a subset of the variables: $\{S_i\} \subseteq \{x_1, \ldots, x_M\}$

- Problem: the output is not non-negative

- The probability for a state of a Markov net is

$$P(x_1, \ldots, x_M) = \frac{1}{Z} \exp f(x_1, \ldots, x_M)$$

  Z normalizes the distribution

- More difficult to interpret: an undirected graph; inference is based on marginalization and conditioning (simpler than in BNs)

Markov Nets and Physics (Boltzmann Distribution)

- Parameter learning is nonmodular due to the normalizer

$$Z = \sum_{x_1,\ldots,x_M} \exp f(x_1, \ldots, x_M)$$

- Z is called the partition function
- $E(x_1, \ldots, x_M) = -f(x_1, \ldots, x_M)$ can be interpreted as an energy
- $F = -\log Z$ is called the Helmholtz free energy
- $S = \sum_{x_1,\ldots,x_M} \log P(x_1, \ldots, x_M) P(x_1, \ldots, x_M)$ is the entropy; then,

$$F = \langle E \rangle - S$$

with the expected energy

$$\langle E \rangle = \sum_{x_1,\ldots,x_M} P(x_1, \ldots, x_M) E(x_1, \ldots, x_M)$$

Graphical Models

- Bayes nets and Markov nets are examples of graphical models
- The assumption was that one needs to explore low-dimensional interactions to model high-dimensional distributions
- This assumption turned out to be wrong

Probabilistic Mixture Models

- A probabilistic mixture model is

$$P(x_1, \ldots, x_M) = \sum_{i=1}^{H} P(i) P(x_1, \ldots, x_M | i)$$

- The component distributions $P(x_1, \ldots, x_M | i)$ are simple: product of eone-dimensional Gaussians or product of Bernoullis
- One can think of component i to represent a hidden (latent) class, mixture component, or cluster
- Parameter learning via EM (Expectation Maximization) to learn $P(i)$ and the parameters in the mixture components
- In contrast to Graphical Models (Bayes, Markov) mixture, here we do not assume any form of locality!
- Used in speech modelling and many other applications but limited in expressibility

Sampling from a Probabilistic Mixture Models

- Let $\kappa_i = P(i)$
- Sample $i^s \sim \vec{\kappa}$
- Sample $x_1^s, \ldots, x_M^s \sim P(x_1, \ldots, x_M | i^s)$

Infinite Mixtures

- $H \to \infty$: We assume an infinite number of latent classes
- Dirichlet Process Mixture Models (DPMMs)

$$P(x_1, \ldots, x_M) = \sum_{i=1}^{\infty} P(i) P(x_1, \ldots, x_M | i)$$

- An interesting step forward; but has technical problems (slow convergence)

Infinite Mixtures with Continuous Latent Variables

- Continuous latent variable model

$$P(x_1, \ldots, x_M) = \int P(\mathbf{h}) P(x_1, \ldots, x_M | \mathbf{h}) d\mathbf{h}$$

- Again, there are an infinite number of latent classes
- We will always assume $P(\mathbf{h}) = \mathcal{N}(\mathbf{h}; 0, \mathbf{I})$
- $\mathbf{h} = (h_1, \ldots, h_{M_{hidd}})^\top$ and $M_{hidd} \leq M$
- We also use $\mathbf{x} = (x_1, \ldots, x_M)^\top$

Sampling from a Probabilistic Mixture Models with Continuous Variables

- Sample $\mathbf{h}^s \sim \mathcal{N}(\mathbf{h}; 0, \mathbf{I})$
- Sample $x_1^s, \ldots, x_M^s \sim P(x_1, \ldots, x_M | \mathbf{h}^s)$

Continuous Latent Variables: Probabilistic PCA

- $\mathbf{x}$ is a **continuous** M-dimensional variable
- Probabilistic PCA uses

$$P(\mathbf{x}|\mathbf{h}) = \mathcal{N}(\mathbf{x}; \mathbf{g}^{gen}(\mathbf{h}), \sigma^2 \mathbf{I})$$

  Given $\mathbf{h}$, we have a ‘naive Bayes” model

- Here, the linear **generator** $\mathbf{g}^{gen}(\mathbf{h}) = \mathbf{V}\mathbf{h}$ maps an element $\mathbf{h} \in \mathbb{R}^{M_{hidd}}$ to a point in $\mathbf{g}^{gen}(\mathbf{h}) \in \mathbb{R}^{M}$; a **generator** is also called a **decoder**
- We can even write explicitly (using our formulas about linear functions of random variables)

$$P(\mathbf{x}) = \mathcal{N}(\mathbf{x}; 0, \mathbf{V}\mathbf{V}^\top + \sigma^2 \mathbf{I})$$

  In PCA, the columns of the $M \times M_{hidd}$ matrix $\mathbf{V}$ are orthogonal

Continuous Nonlinear Latent Variables

- We simply model $g^{gen}(\mathbf{h})$ by a deep neural network (DNN) with M_{hidd} inputs and M outputs!
- $\mathbf{g}^{gen}(\cdot)$ forms an M_{hidd}-dimensional manifold in $\mathbb{R}^M$
- **This is the basis for modern DNN-based generative models!**
- Problem: there is no convenient expression for

$$P(\mathbf{x}) = \int P(\mathbf{x}|\mathbf{h})P(\mathbf{h})d\mathbf{h}$$

Continuous Nonlinear Latent Variables without the Noise

- With $\sigma^2 \rightarrow 0$

$$P(\mathbf{x}|\mathbf{h}) = \delta(\mathbf{x} - \mathbf{g}^{gen}(\mathbf{h}))$$

- $\delta((\mathbf{x})$ is the Dirac delta function: a delta-peak where the argument is equal to 0
- Only the $x_1, \ldots, x_M$ on the manifold obtain nonzero probability!
- We now have a function

$$\mathbf{x}(\mathbf{h}) = \mathbf{g}^{gen}(\mathbf{h})$$

- This is used in the autoencoder (AE) and in GANs

Invertible Maps

- If $M_{hidd} = M$ some generators are invertible $\mathbf{g}^{gen}(\mathbf{g}^{enc}(\mathbf{x})) = \mathbf{x}$
- $\mathbf{g}^{enc}$ is called the **encoder**

$$P(\mathbf{x}) =$$

$$|\det \mathbf{J}(\mathbf{x})| \times \mathcal{N}(\mathbf{g}^{enc}(\mathbf{x}); 0, \mathbf{I})$$

- Here, $\mathbf{J}$ is the Jacobian matrix with $J_{i,j}(\mathbf{x}) = \frac{\partial h_i}{\partial x_j} = \frac{\partial g_i^{enc}(\mathbf{x})}{\partial x_j}$.
- This is the basis for normalizing flow (e.g., flow matching, we will not cover)

E-step

- With noise $\sigma^2 > 0$, any $\mathbf{x}$ can be generated (some might have tiny probabilities) and we can calculate the probabilities of the latent classes using Bayes formula

$$P(\mathbf{h}|\mathbf{x}) = \frac{P(\mathbf{x}|\mathbf{h})P(\mathbf{h})}{P(\mathbf{x})}$$

  This is required in the E-step; one problem now is that $P(\mathbf{h}|\mathbf{x})$ can be highly complex

Encoder

- With no noise $\sigma^2 = 0$, and with $M_{hidd} < M$, the generator is not invertible
- For almost all $\mathbf{x}$, $P(\mathbf{x}) = 0$
- An **encoder** $\mathbf{h} = \mathbf{g}^{enc}(\mathbf{x})$ returns the $\mathbf{h}$ such that

$$\mathbf{h}^{pi} = \arg\min \|\mathbf{x} - \mathbf{g}^{gen}(\mathbf{h})\|^2$$

  is minimum; it is the pseudo-inverse of the generator
- We can evaluate this distance on a subset of known dimensions of $\mathbf{x}$ and then define $\hat{\mathbf{x}} = g^{gen}(\mathbf{h}^{pi})$
- $\hat{\mathbf{x}}$ is now a complete vector: useful for recommendation systems and inpainting!

Sampling (Generating Images) is Very Simple

- Generating samples is easy
- First, generate a sample as $\mathbf{h} \sim \mathcal{N}(\mathbf{h}; 0, \mathbf{I})$
- And then $\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h})$
- This is the basis for generative AI

Conditional Generators

- We can learn class-specific generators

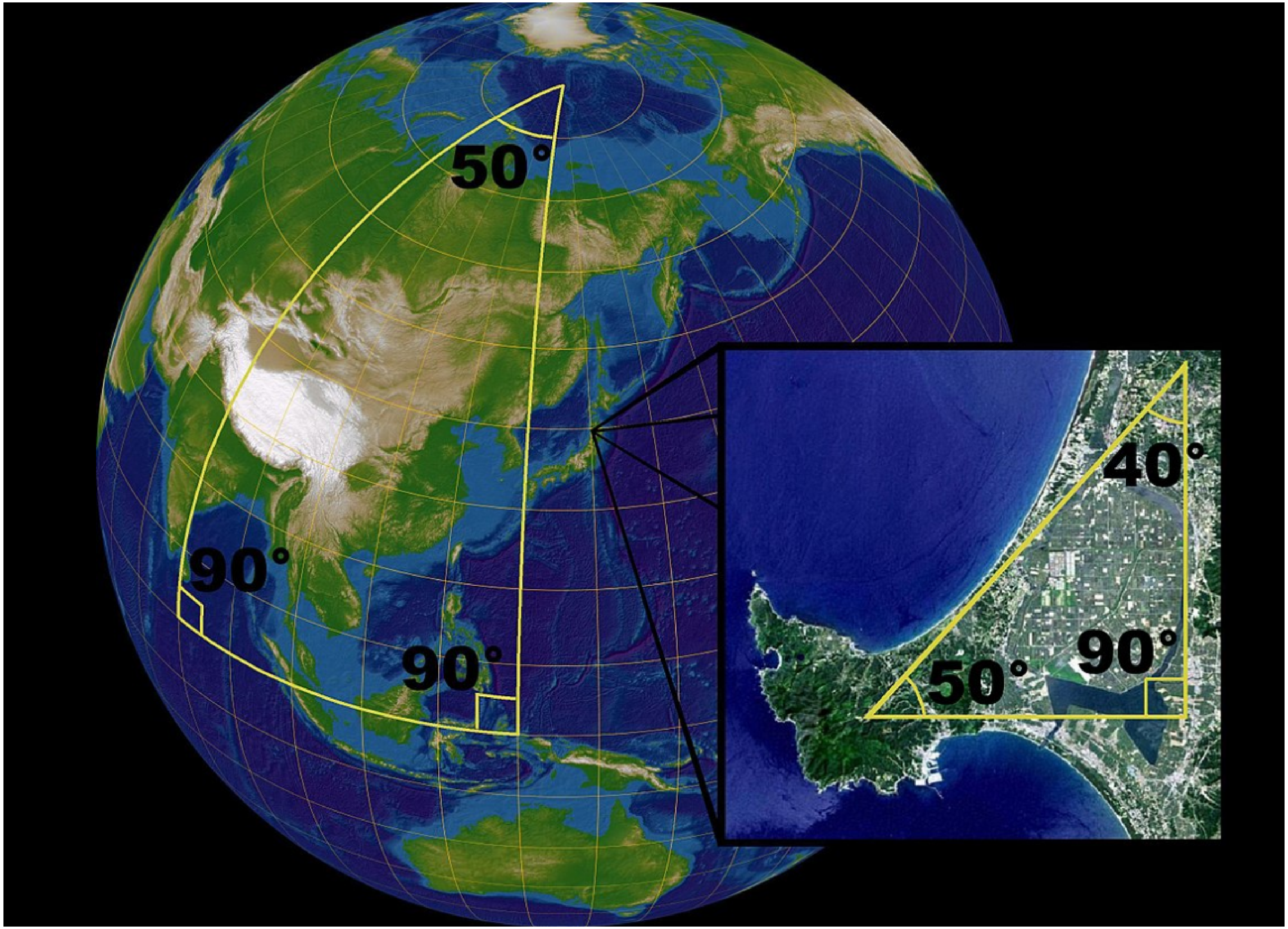
$$\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h}, m_1, \ldots, m_{M_m})$$

- In recommendation systems, $m_1, \ldots, m_{M_m}$ can be user attributes (young, male, ...)
- In generative AI for vision, the can specify the desired generated image (Cat, Black, ...)

Part II

Manifolds

- In mathematics, a manifold is a topological space that locally resembles Euclidean space near each point
- A topological space may be defined as a set of points, along with a set of neighbourhoods for each point, satisfying a set of axioms relating points and neighbourhoods


50°
90°
90°
40°
90°
50°

Data Represented in Feature Space

- Consider Case Ib (manifold): input data only occupies a manifold
- Example: consider that the data consists of face images; all images that look like faces would be part of a manifold
- What is a good mathematical model? We assume that “nature” produces data in some low-dimensional space $\mathbf{h}^{nat} \in \mathbb{R}^H$, but nature only makes data available in some high-dimensional feature space $\mathbf{x} \in \mathbb{R}^M$ ($\mathbf{x}$ might describe an image, in which case M might be a million)
- Features map

$$\mathbf{x} = \mathsf{featureMap}(\mathbf{h}^{nat})$$

featureMap is the generator used by nature

Manifold in Machine Learning

- In Machine Learning: in the observed M-dimensional input space, the data is distributed on an M_{hidd}-dimensional manifold

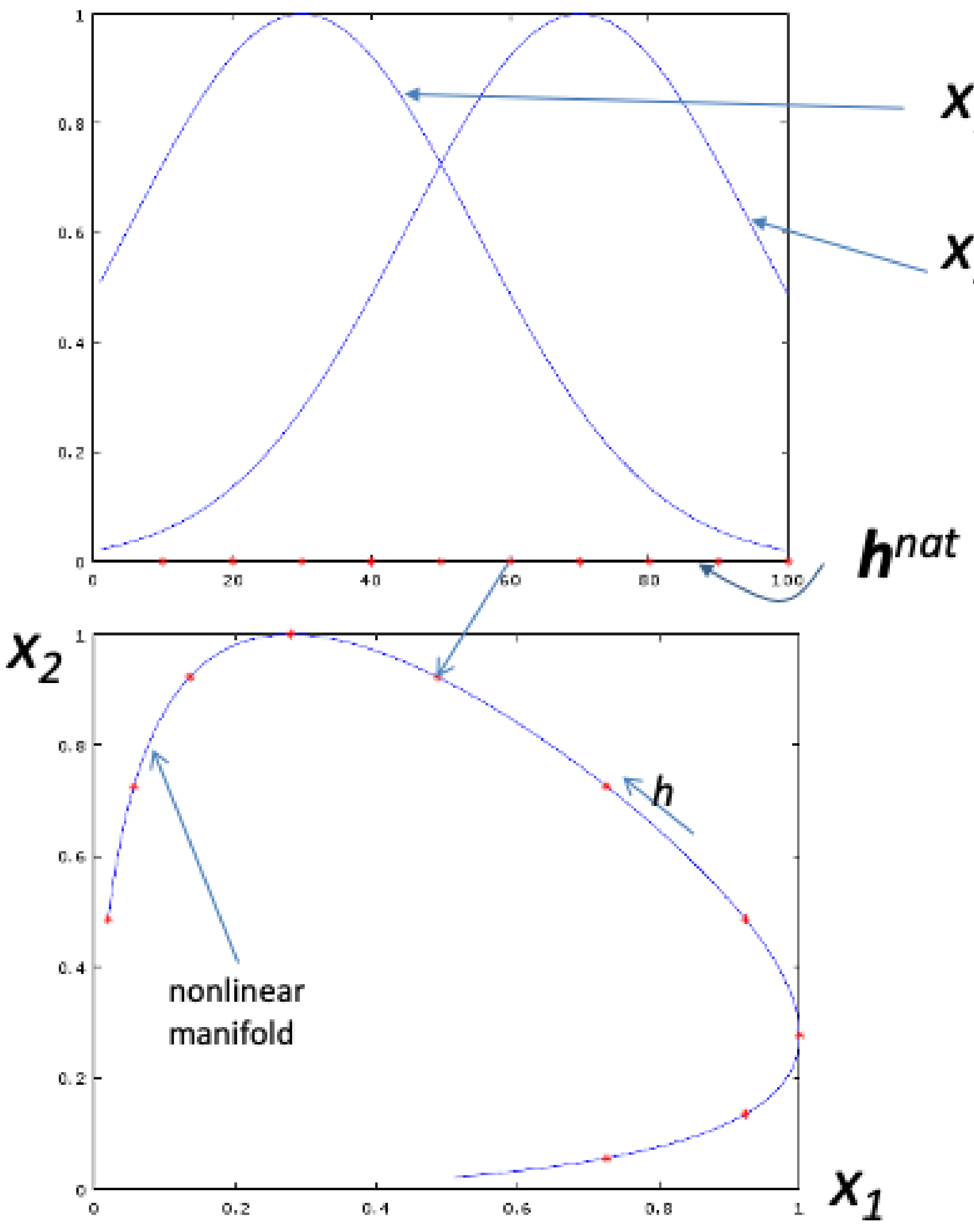
$$\{\mathbf{x} \in \mathbb{R}^M : \exists \mathbf{h} \in \mathbb{R}^H \ \ s.th. \ \ \mathbf{x} = \mathbf{g}^{gen}(\mathbf{h})\}$$

  where $\mathbf{g}^{gen}(\cdot)$ is smooth

- Note that for a given $\mathbf{x}$, it is not easy to see if it is on the manifold

$x_1 = featureMap_1(\mathbf{h}^{nat})$

$x_2 = featureMap_2(\mathbf{h}^{nat})$



$\mathbf{h}^{nat}$ is not measured: it is latent

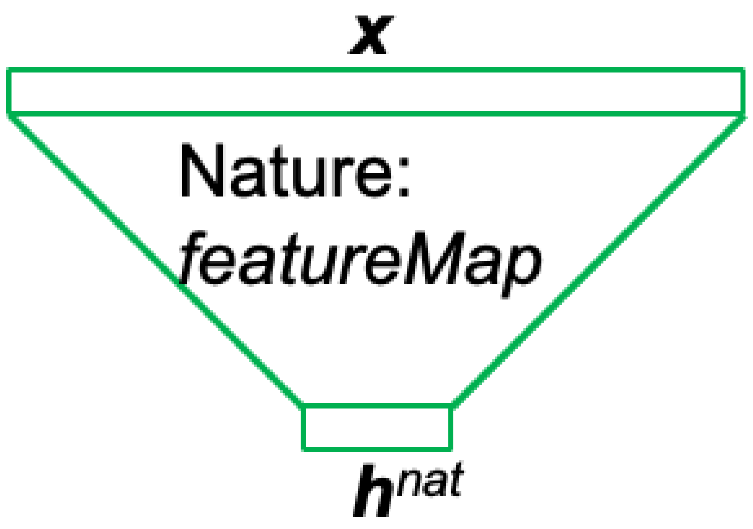
here: $M_{hidd}=1$

The data is available in feature space $\mathbf{x}$

here: $M=2$

Feature Engineering

- In a way, features are like basis functions, but supplied by nature or an application expert (feature engineering)
- In the spirit of the discussion in the lecture on the Perceptron: $\mathbf{h}^{nat}$ might be low-dimensional and explainable, but we can only measure $\mathbf{x}$



The feature map produces an M-dimensional observed $\boldsymbol{x}$; but the data only occupies an M_{hidd}-dimensional manifold

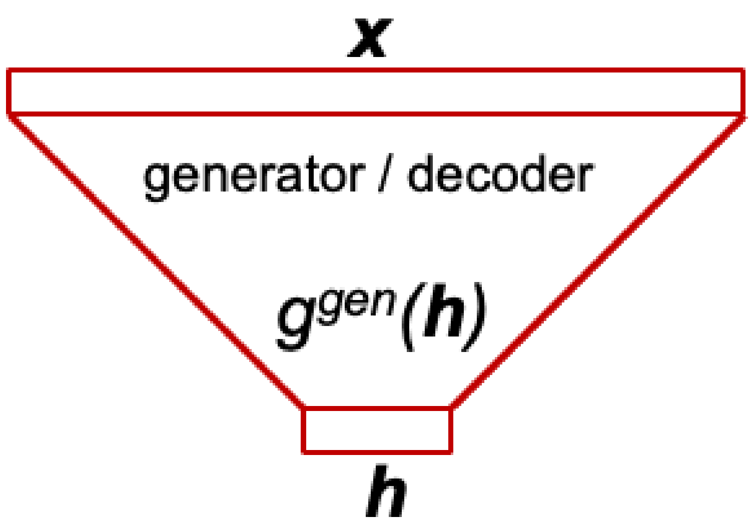
Nature selects an instance of a low dimensional $\boldsymbol{h}^{nat}$ M_{hidd}-dimensional

Learning a Generator / Decoder

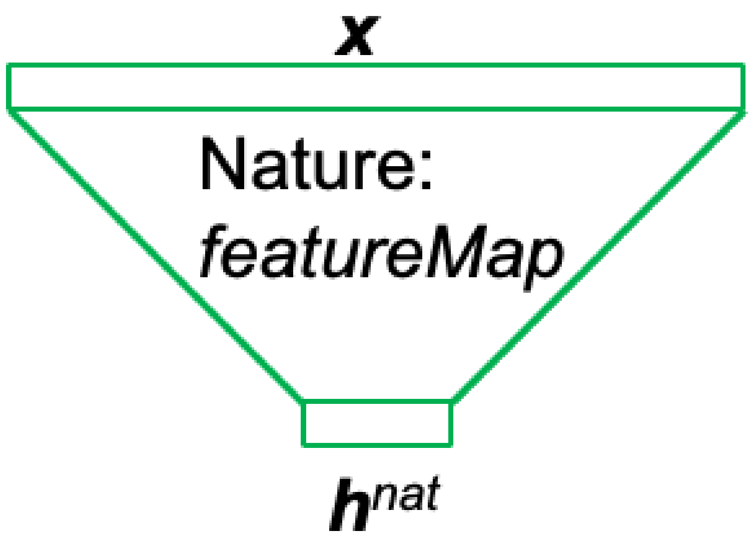
- Example: $\mathbf{x}$ represents a face; let's assume nature selects $\mathbf{h}^{nat}$ from an M_{hidd}-dimensional Gaussian distribution with unit covariance
- Then, if the feature map is known, we can generate new natural looking faces of people who do not exist: $\mathbf{x} = \text{featureMap}(\mathbf{h}^{nat})$
- We try to emulate this process by a model

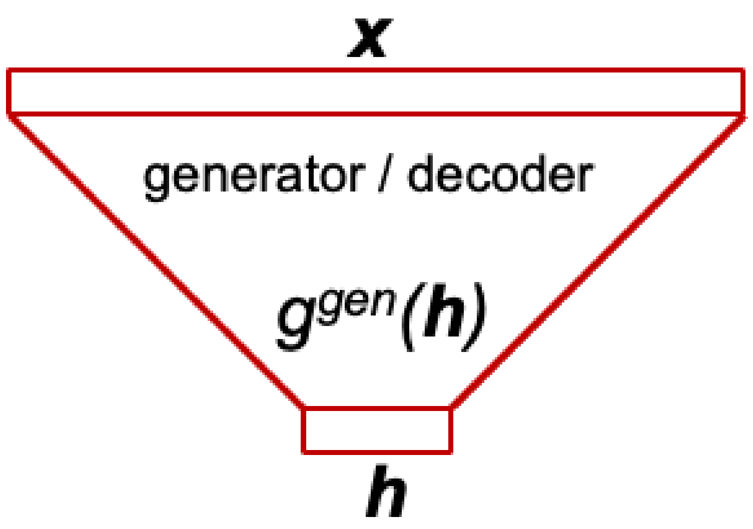
$$\mathbf{x} = g^{gen}(\mathbf{h})$$

  (here we drop the superscript nat, since this $\mathbf{h}$ is part of our model and not the ground truth $\mathbf{h}^{nat}$)
- In an autoencoder, the **generator** is also called a **decoder**



- A generator tries to copy this generative process
- If the $\boldsymbol{x}$ represent images, one can now generate new natural looking images





- A generator maps from M_h-dimensional $\boldsymbol{h}$ to an M-dimensional $\boldsymbol{x}$
- $M_h \ll M$
- Any $\boldsymbol{h}$ gives a valid image $\boldsymbol{x}$

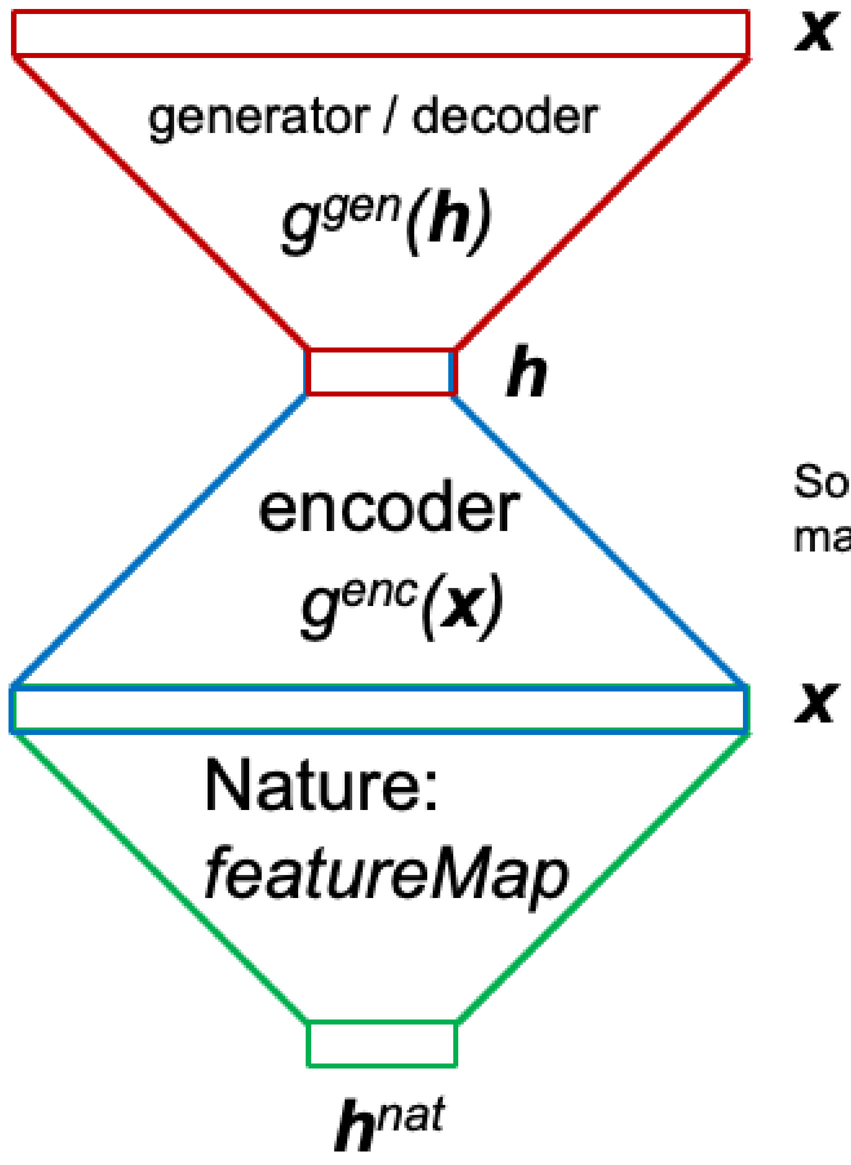
- In the simplest case, this is a map between two layers in a neural network
- In practice, the map is described by a DNN

- If we assume that the $\boldsymbol{h}$ are generated randomly, then the dimensions in $\boldsymbol{x}$ are strongly dependent (highly correlated)

Learning an Encoder

- Of course for data point i we only know $\mathbf{x}_i$ but we do not know $\mathbf{h}_i^{nat}$
- But maybe we can estimate $\mathbf{h}_i \approx \mathbf{h}_i^{nat}$ based on some **encoder** neural network

$$\mathbf{h}_i = \mathbf{g}^{enc}(\mathbf{x}_i)$$



Some models contain an explicit encoder with maps $\boldsymbol{x}$ to a latent repesentation

(only $\boldsymbol{x}$ can be measured)

Encoder

- An encoder can be useful by itself: it can serve as a preprocessing step in a classification problem (Case Ib (manifold))

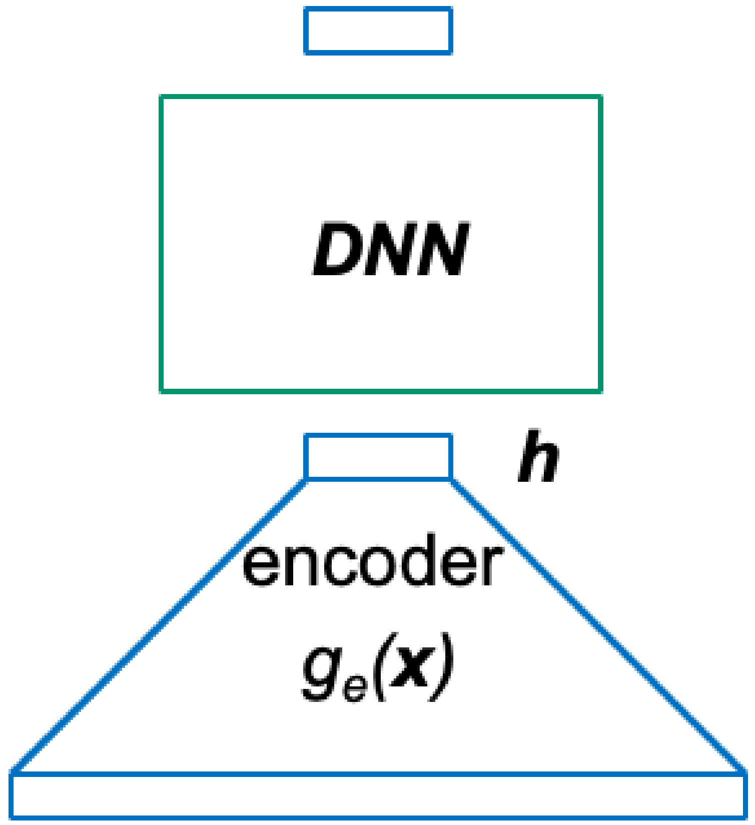
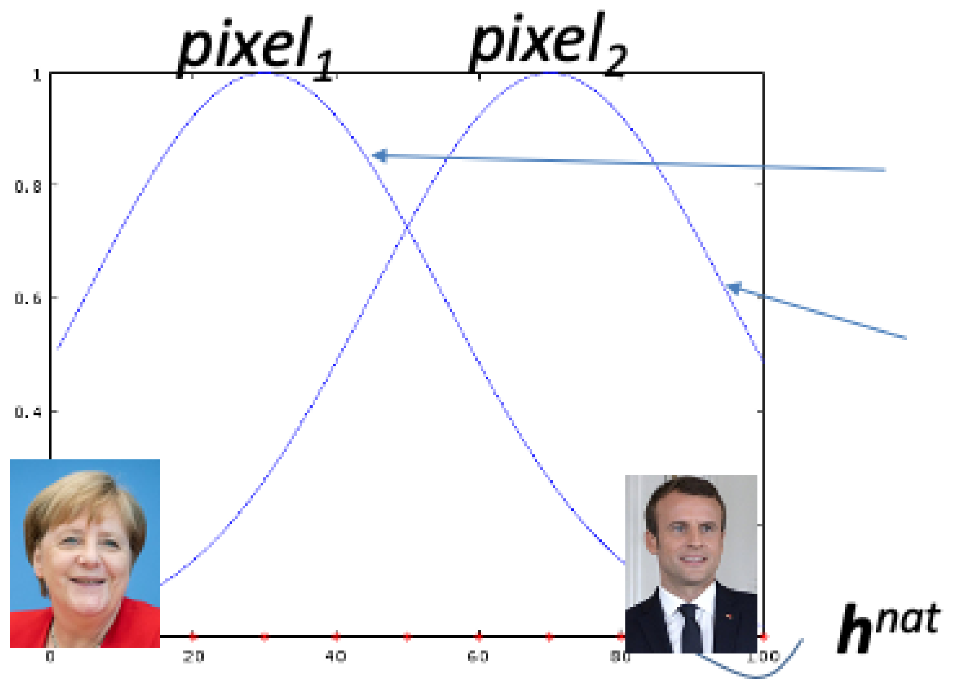


Image class

A trained encoder can be used as a preprocessing step in classification/regression

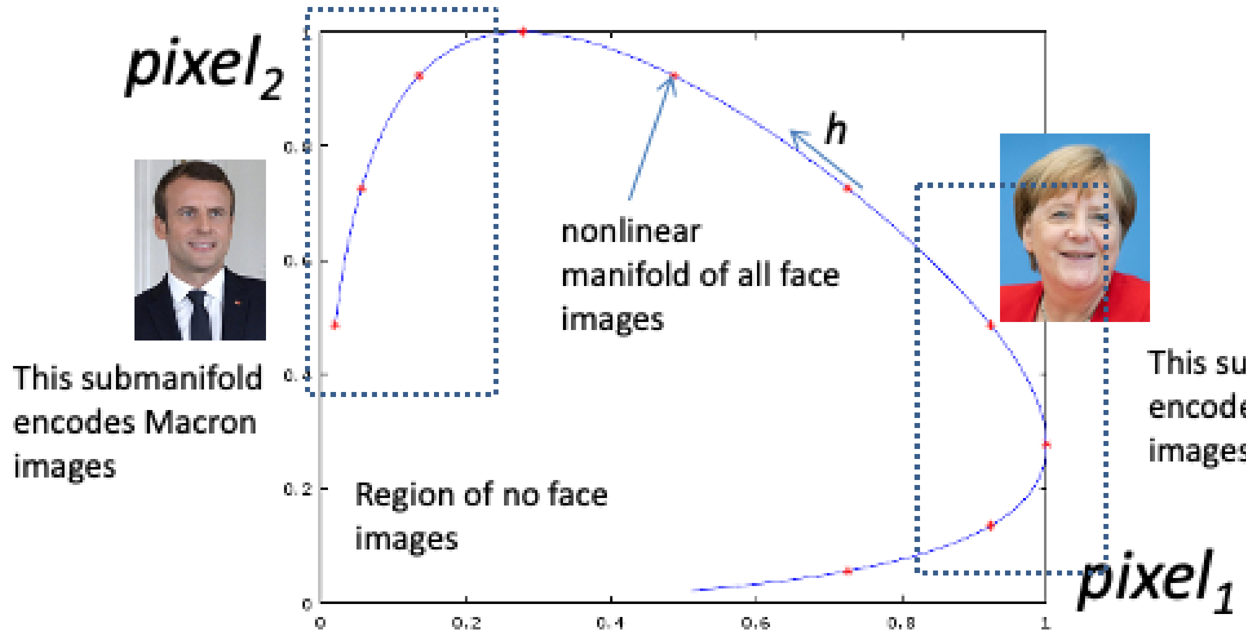
$\boldsymbol{x}$ (only $\boldsymbol{x}$ can be measured)



Generator/Decoder:

generates face images for any ***h***, e.g., of Merkel and Marcon (and others)

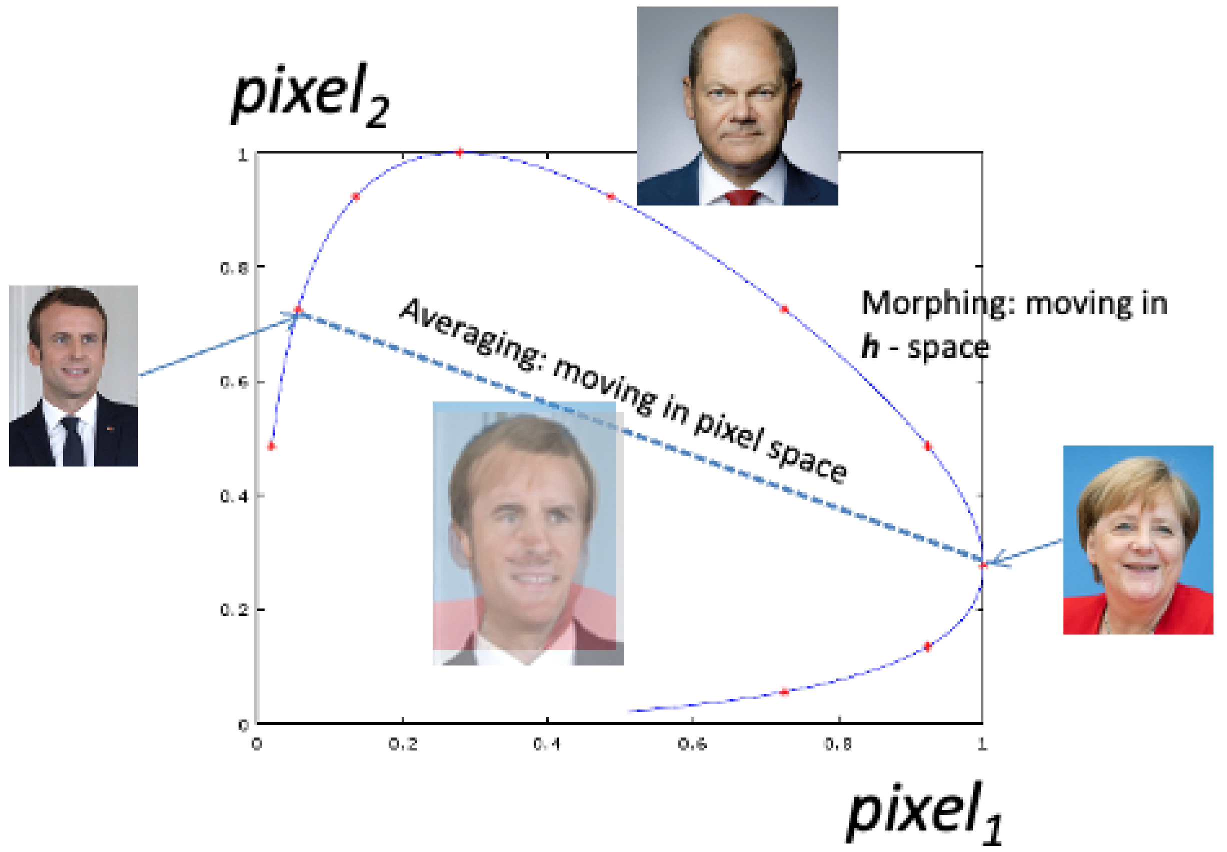
$$x = g^{gen}(h)$$



Encoder:

calculates embeddings if ***x*** is on the manifold, i.e., is a valid face image

$$h = g^{enc}(x)$$



- *If we move from Merkel to Macron in pixel space (**x**) in a straight line, we get averaged faces in between*
- *If we move from Merkel to Macron in latent space (**h**) in a straight line and then apply the generator* $\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h})$ *, we get valid sharp faces in between, maybe the face of Scholz*

Learning an Autoencoder

Learning an Encoder

- If we have,

$$\mathbf{x} = \text{featureMap}(\mathbf{h}^{nat})$$

  we might want to learn an approximate inverse of the feature map

$$\mathbf{h} = \mathbf{g}^{enc}(\mathbf{x})$$

- $\mathbf{g}^{enc}(\mathbf{x})$ is called an encoder

Autoencoder

- How can we learn $\mathbf{h} = \mathbf{g}^{enc}(\mathbf{x})$ if we do not measure $\mathbf{h}$?
- Consider a decoder (which might be close to the feature map)

$$\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h})$$

  But again, $\mathbf{h}$ is not measured

- We now simply concatenate the two models and get

$$\hat{\mathbf{x}} = \mathbf{g}^{gen}(\mathbf{g}^{enc}(\mathbf{x}))$$

- This is called an autoencoder

Linear Autoencoder

- If the encoder and the decoder are linear functions, we get a linear autoencoder
- A special solution is provided by the **principal component analysis** (PCA)

  Encoder:

$$\mathbf{h} = \mathbf{g}^{enc}(\mathbf{x}) = \mathbf{V}_h^T \mathbf{x}$$

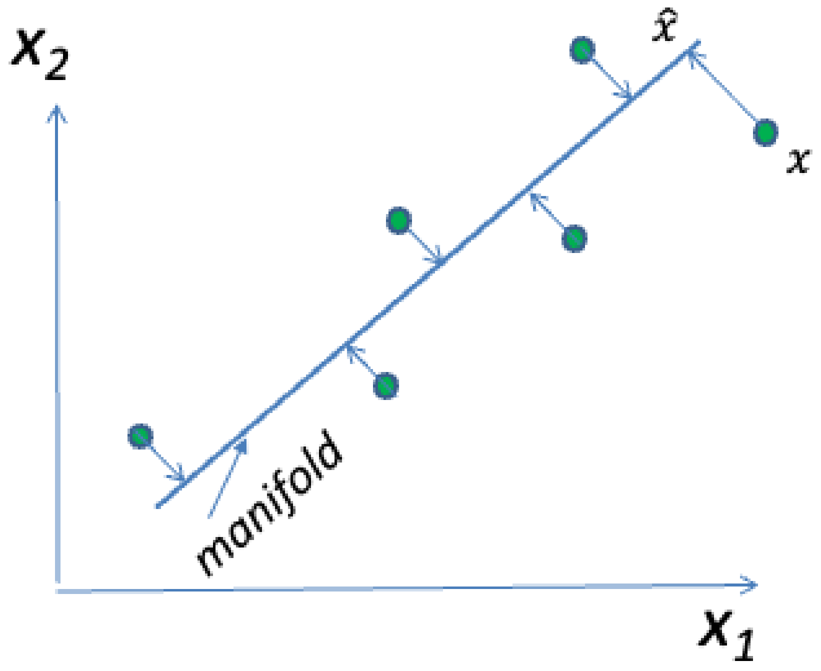
  Decoder:

$$\hat{\mathbf{x}} = \mathbf{g}^{gen}(\mathbf{h}) = \mathbf{V}_H \mathbf{h} = \mathbf{V}_H \mathbf{V}_H^T \mathbf{x}$$

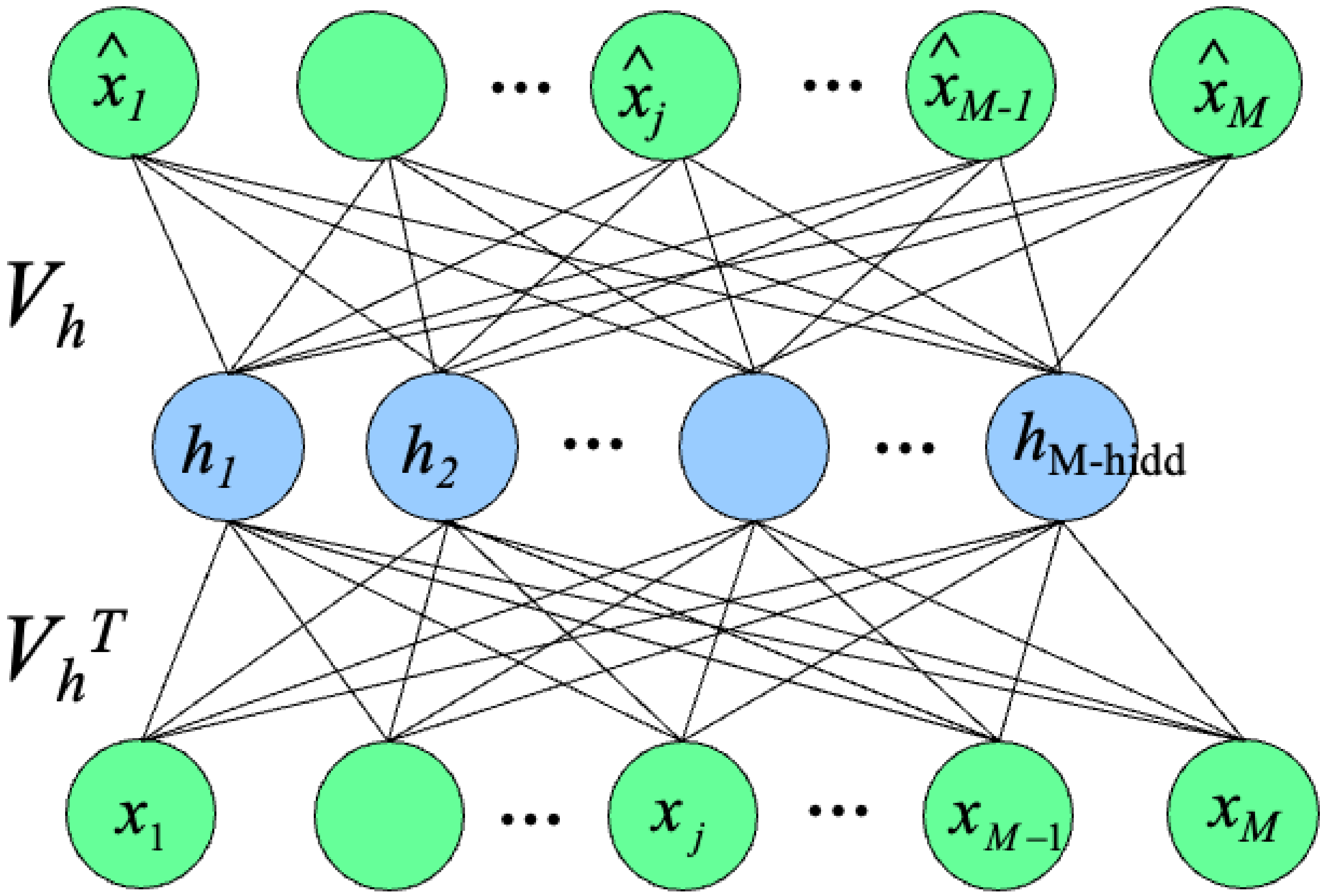
- The $\mathbf{V}_H$ are the first M_{hidd} columns of $\mathbf{V}$, where $\mathbf{V}$ is obtained from the **singular value decomposition** SVD

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

$$\hat{x} = V_h V_h^T x$$



The manifold is a linear subspace



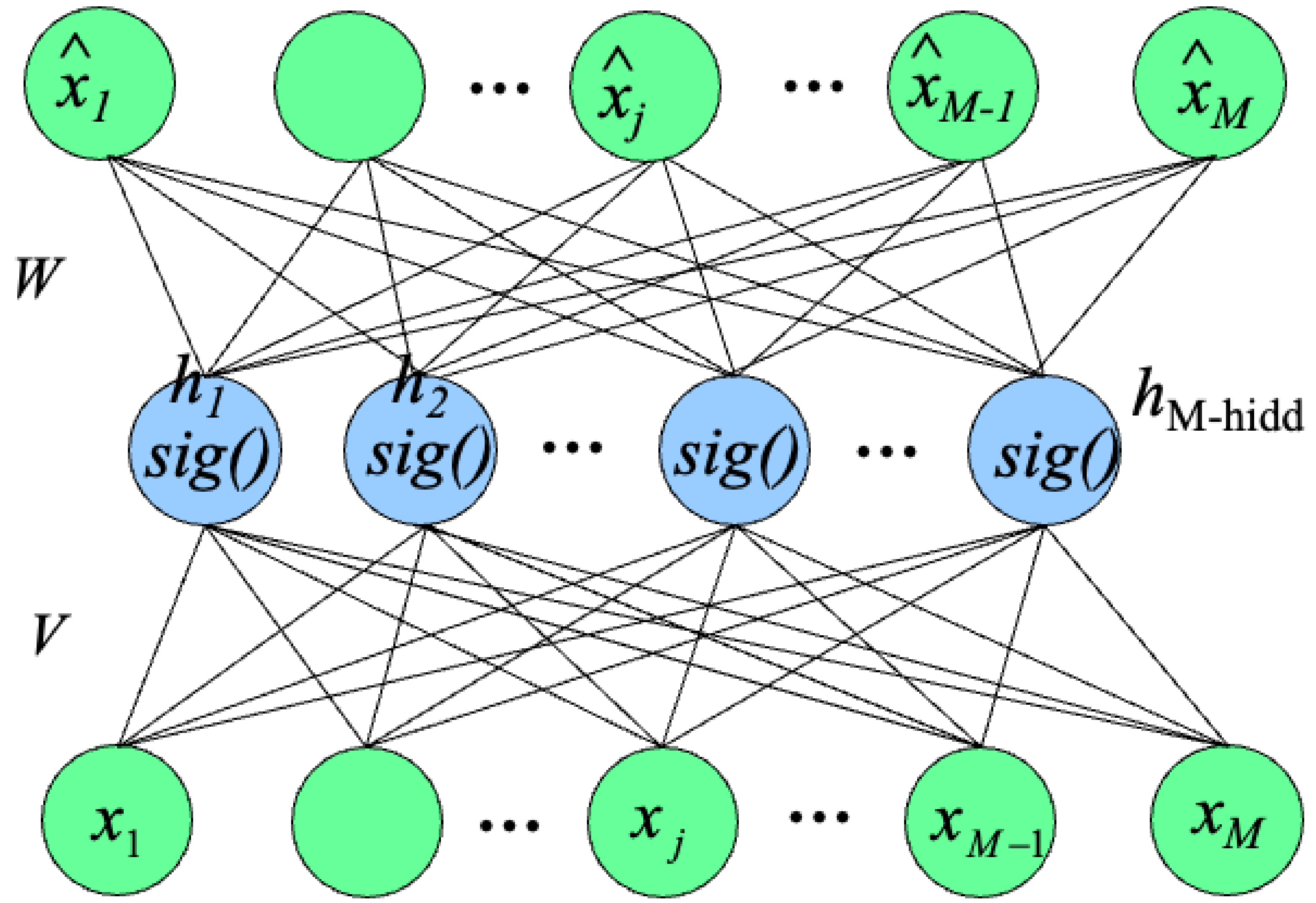
$\hat{x}_1$
...
$\hat{x}_j$
...
$\hat{x}_{M-1}$
$\hat{x}_M$
V_h
h_1
h_2
...
...
$h_{\text{M-hidd}}$
V_h^T
x_1
...
x_j
...
x_{M-1}
x_M

Neural Network Autoencoder

- In the Neural Network Autoencoder, the encoder and the decoder are modelled by neural networks
- The cost function is

$$\text{cost}(W, V) = \sum_{i=1}^{N} \sum_{j=1}^{M} (x_{i,j} - \widehat{x}_{i,j})^2$$

where $\widehat{x}_{i,1}, \ldots, \widehat{x}_{i,M}$ are the outputs of the neural network autoencoder


$\hat{x}_1$
...
$\hat{x}_j$
...
$\hat{x}_{M-1}$
$\hat{x}_M$
W
h_1
h_2
$h_{\text{M-hidd}}$
sig()
sig()
...
sig()
...
sig()
V
x_1
...
x_j
...
x_{M-1}
x_M

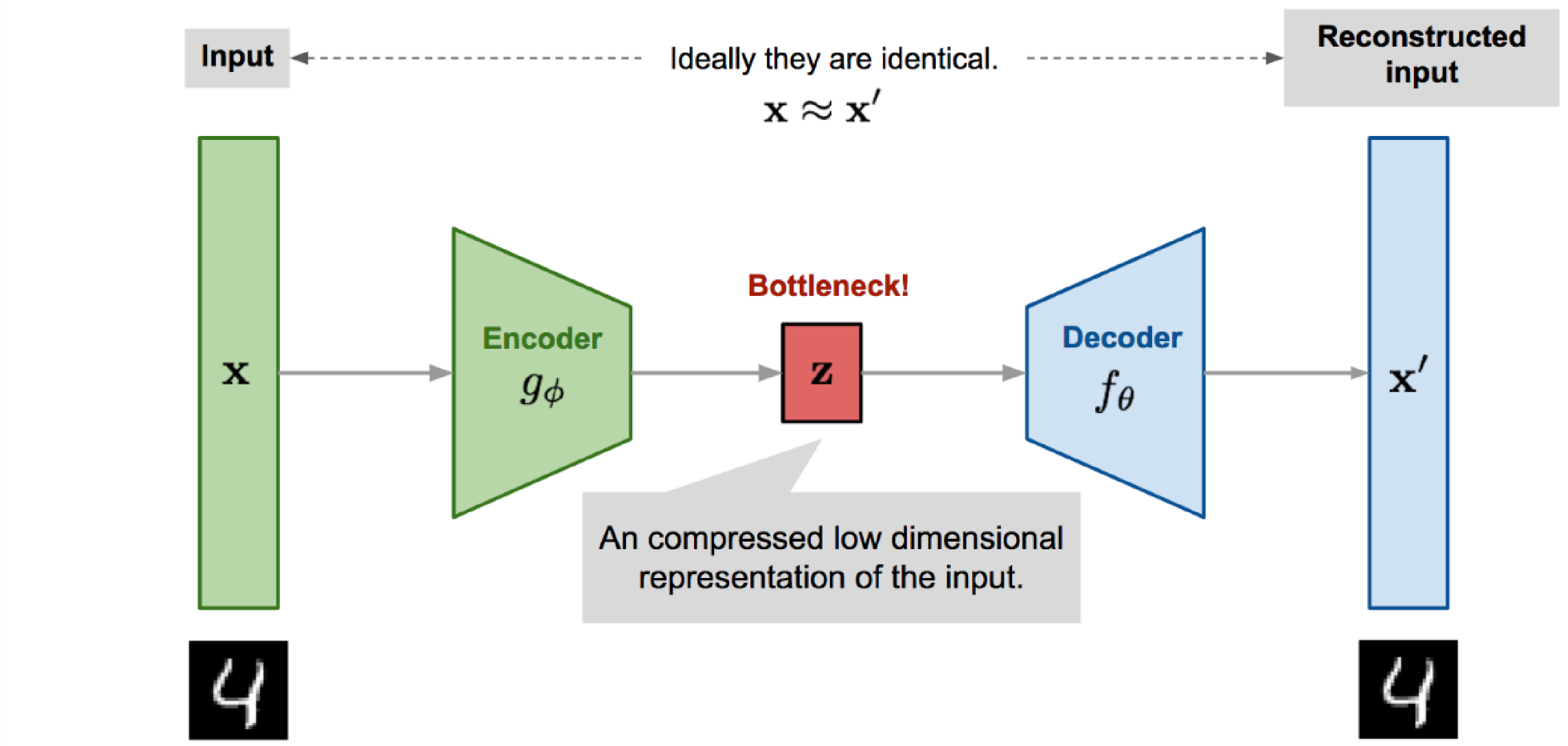
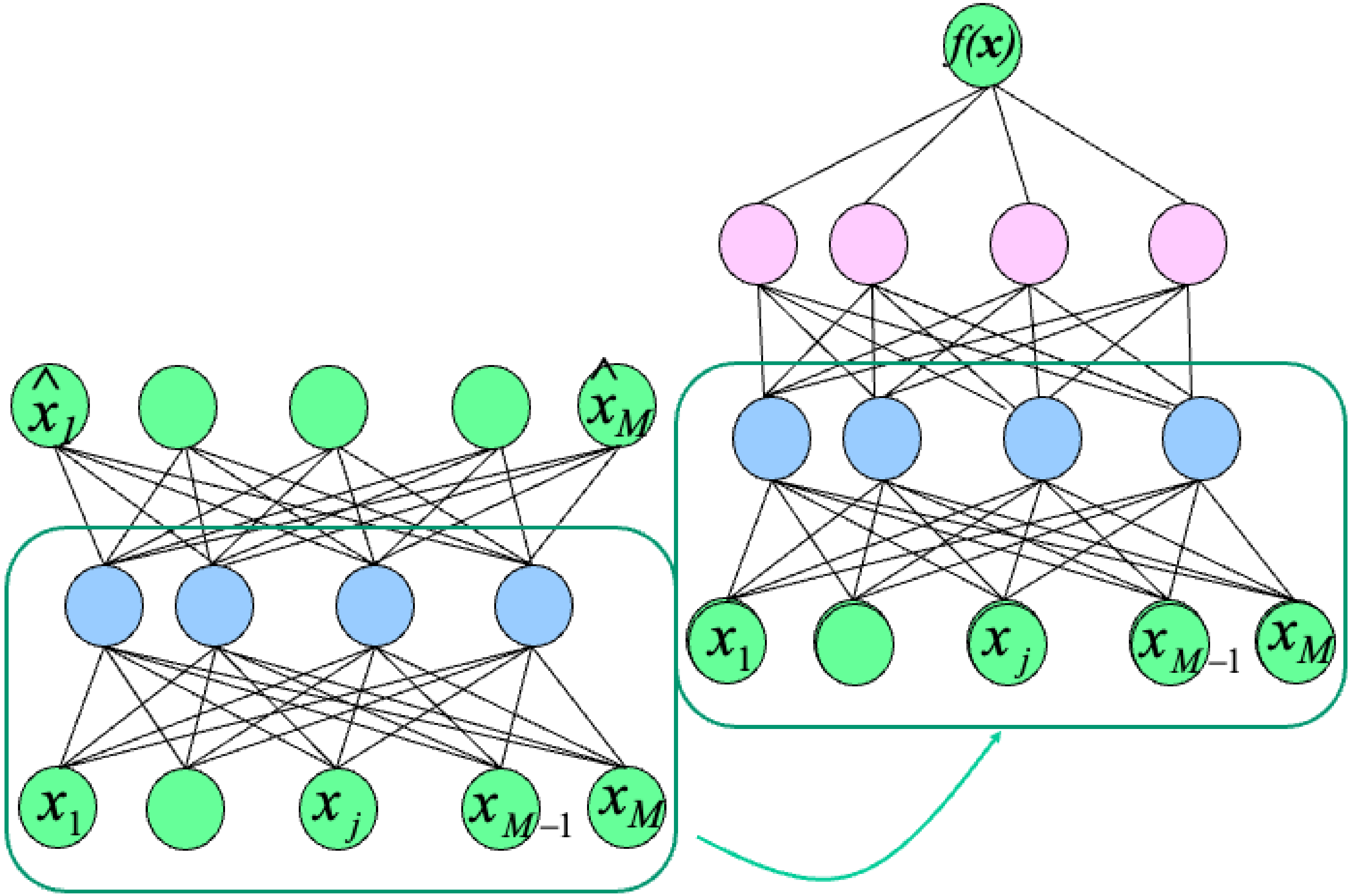

Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
$\mathbf{x}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
Decoder
f_θ
$\mathbf{x}'$
An compressed low dimensional representation of the input.

Fig. 1. Illustration of autoencoder model architecture.

Comments and Applications

- Since $\mathbf{h}$ cannot directly be measured, it is called a latent vector in a latent space. The representation of a data point $\mathbf{x}_i$ in latent space $\mathbf{h}_i$ is called its representation or embedding
- Distances in latent space are often more meaningful than in data space, so the latent representations can be used in information retrieval
- The reconstruction error $||\mathbf{x} - \hat{\mathbf{x}}||^2$ is often large for patterns which are very different from the training data; the error thus measures novelty, anomality. This can be a basis for fraud detection and plant condition monitoring
- The encoder can be used to pretrain the first layer in a neural network; after initialization, the complete network is then typically trained with backpropagation, including the pretrained layer


$f(\boldsymbol{x})$
$\hat{x}_1$
$\hat{x}_M$
x_1
x_j
x_{M-1}
x_M
x_1
x_j
x_{M-1}
x_M

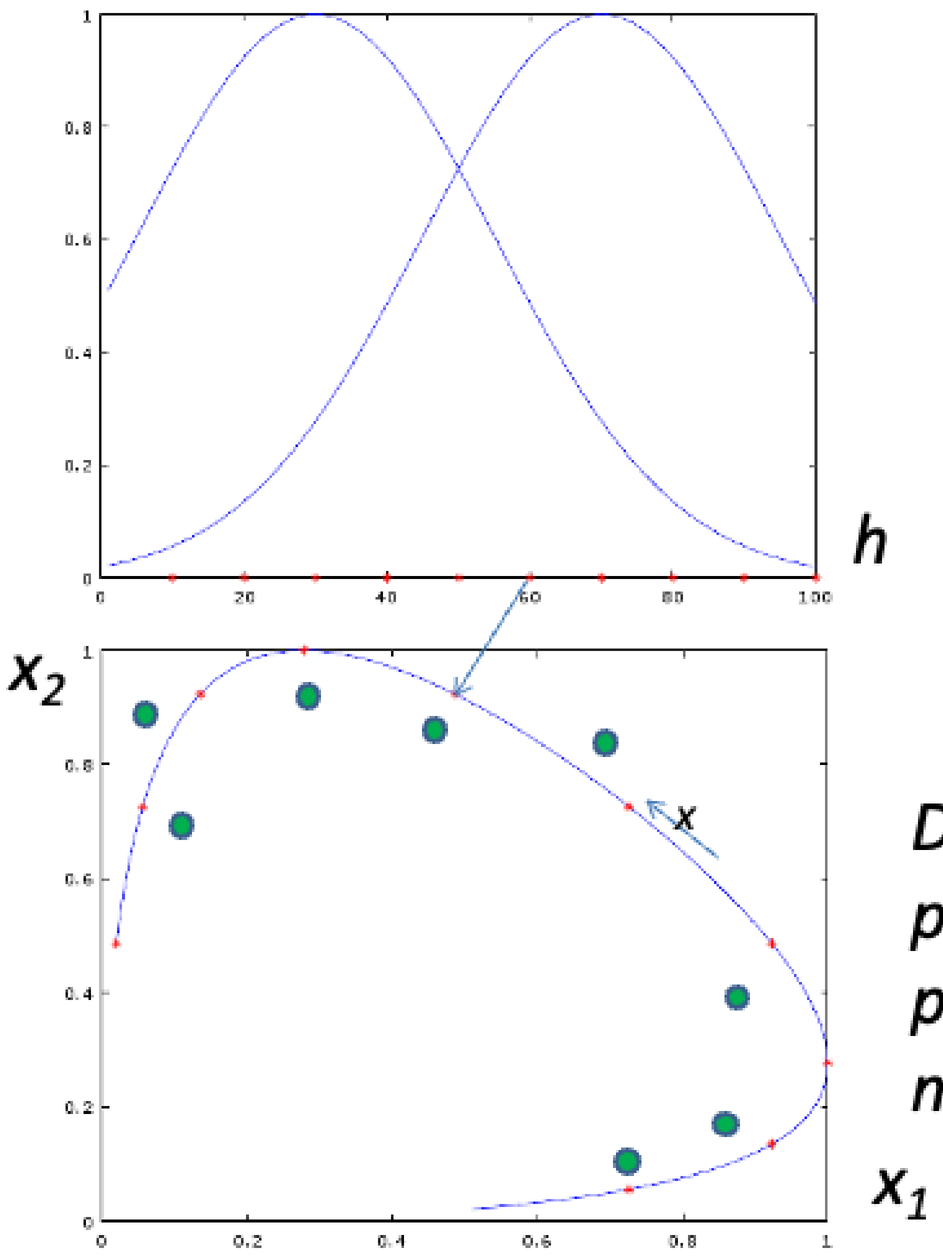
Data Represented in a Noisy Feature Space

- The feature map might include some noise,

$$\mathbf{x} = \text{featureMap}(\mathbf{h}^{nat}) + \vec{\epsilon}$$

  where $\vec{\epsilon}$ is a noise vectors; then the data might only be exactly on the manifold

- One would want that $\mathbf{g}^{enc}(\text{featureMap}(\mathbf{h}^{nat}) + \vec{\epsilon}) \approx \mathbf{g}^{enc}(\text{featureMap}(\mathbf{h}^{nat}))$ such that the encoder is noise insensitive; this is enforced by the **denoising autoencoder**



Due to some noise process, the data points do not lie on the manifold

Denoising Autoencoder (DAE)

- Denoising autoencoder,

$$\mathbf{x}_i \leftarrow \mathbf{g}^{gen}(\mathbf{g}^{enc}(\mathbf{x}_i + \vec{\epsilon}_i))$$

  where $\vec{\epsilon}_i$ is random noise added to the input!

- This also prevents an autoencoder from learning an identity function

- The "noise" does not have to be additive and distortions can assume many forms (e.g., masking pixels)

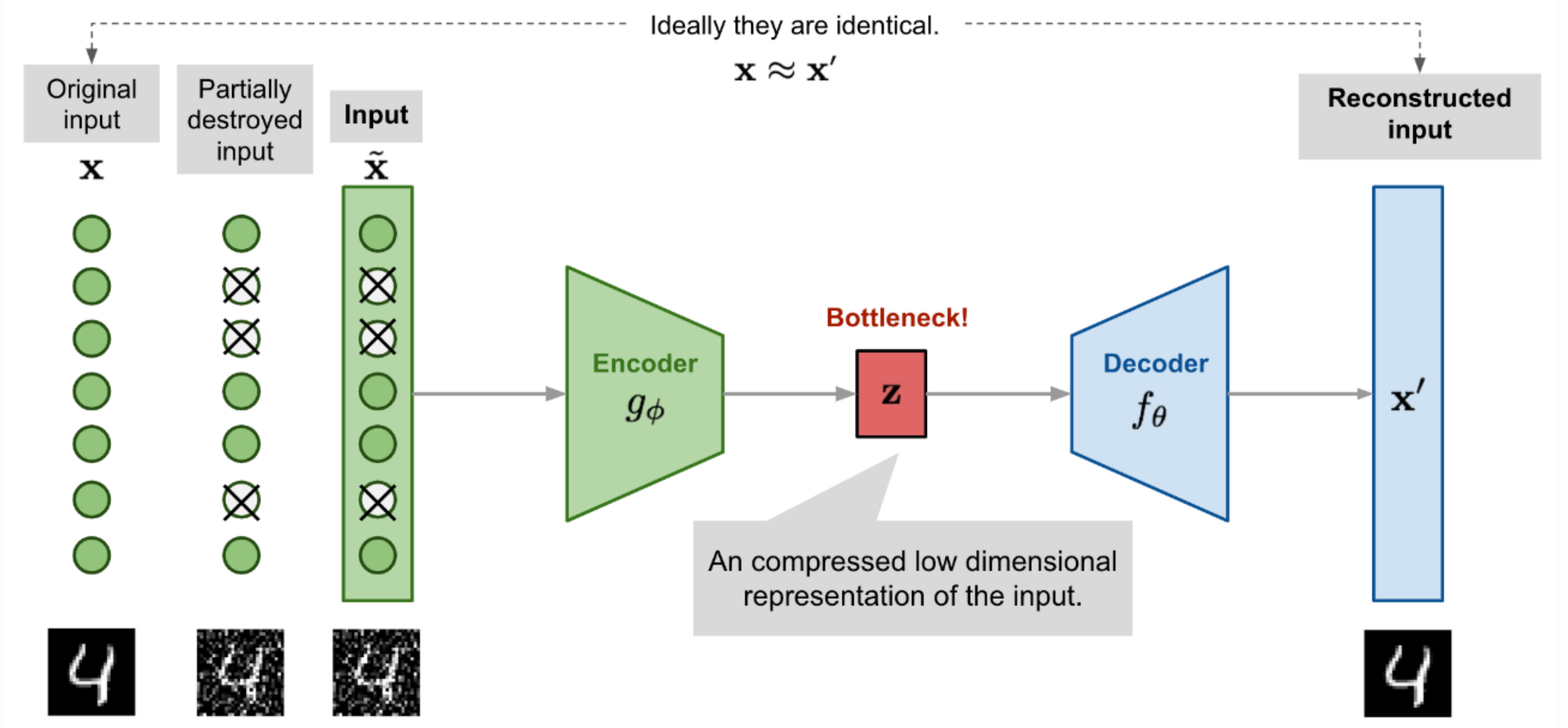

Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Original input
$\mathbf{x}$
Partially destroyed input
Input
$\tilde{\mathbf{x}}$
Encoder
g_ϕ
Bottleneck!
$\mathbf{z}$
An compressed low dimensional representation of the input.
Decoder
f_θ
Reconstructed input
$\mathbf{x}'$

Fig. 2. Illustration of denoising autoencoder model architecture.

Learning a Generator

Learning a Decoder (Generator)

- Requirement 1: No manifold in $\mathbf{h}$-space: any random $\mathbf{h}$

$$\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h})$$

  generates a high-quality image

- Requirement 2: Smoothness: Two vectors which are close in $\mathbf{h}$ space should produce similar images (otherwise the map to $\mathbf{x}$ would be impossible to learn)

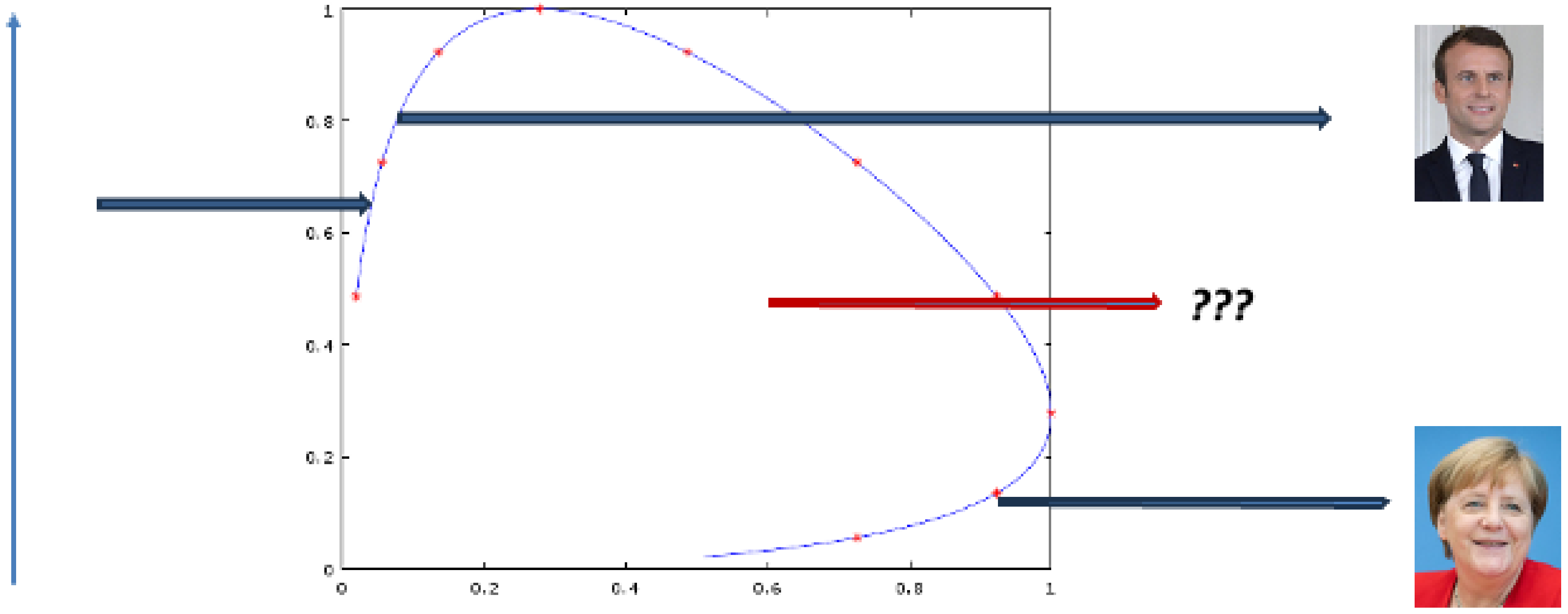
- Requirement 3: Disentanglement: $h_1, \ldots, h_H$ can be interpreted

- Requirement 4: Conditional (attribute specific) models

$$\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h}, \mathbf{m}) \qquad \mathbf{h} = \mathbf{g}^{enc}(\mathbf{x}, \mathbf{m})$$

  The generated image has real features $\mathbf{m} = (m_1, \ldots, m_{M_m})^\top$; do I want a smiling face, do I want a beard, glasses, sunglasses, ...?

- One could argue that the dimension of $\mathbf{h}$ should now be $H - M_m$



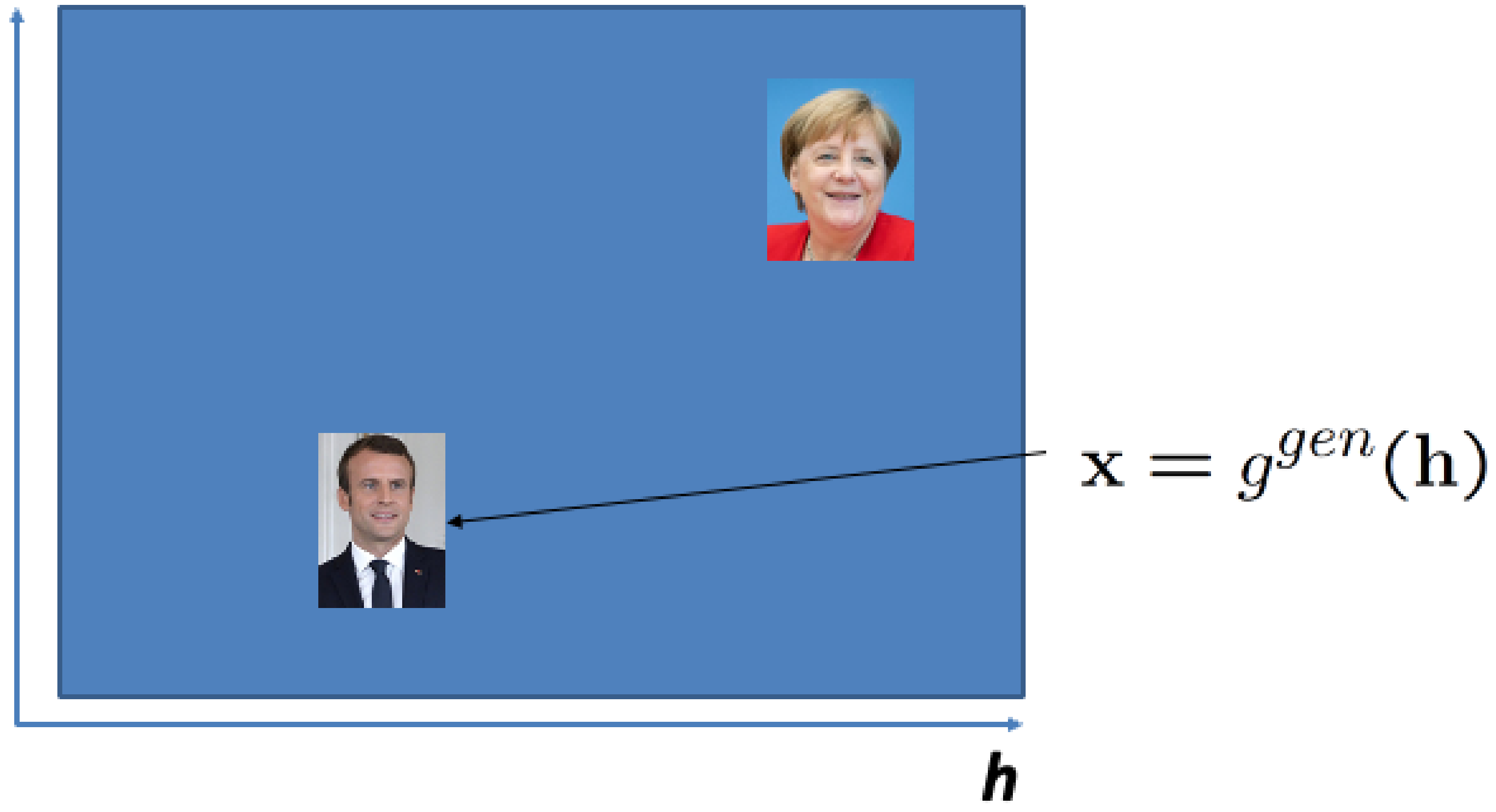
h:
Here a 1-D ***h*** is optimal:
every ***h*** gives a valid face image

h:
The model might have generated a 2-D latent space:
Only the 1-D manifold in the 2-D space ***h*** on the manifold h gives a valid face image

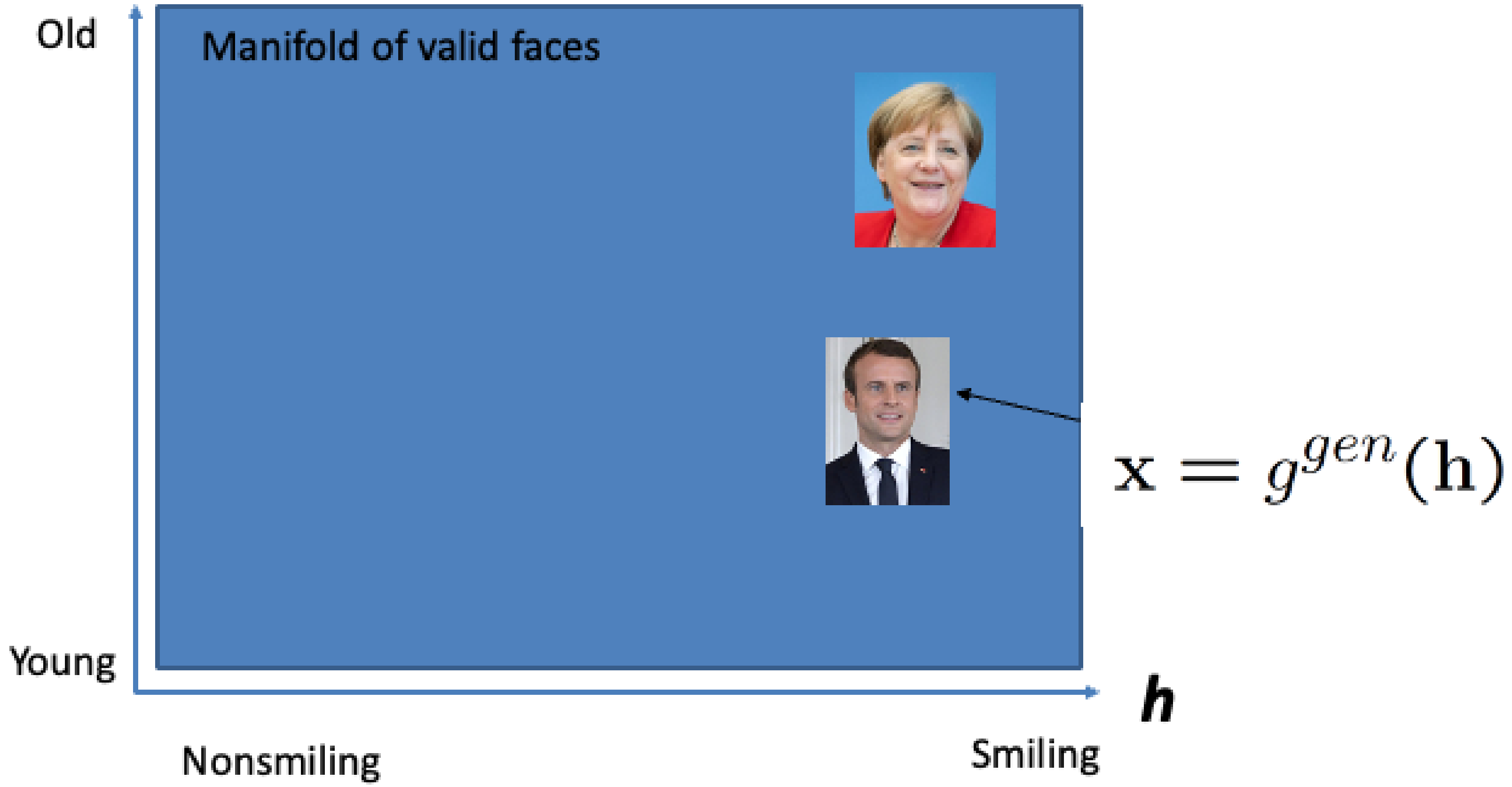
x

Requirement 1: no manifold

In the Right Space, the Whole Space Maps to Faces

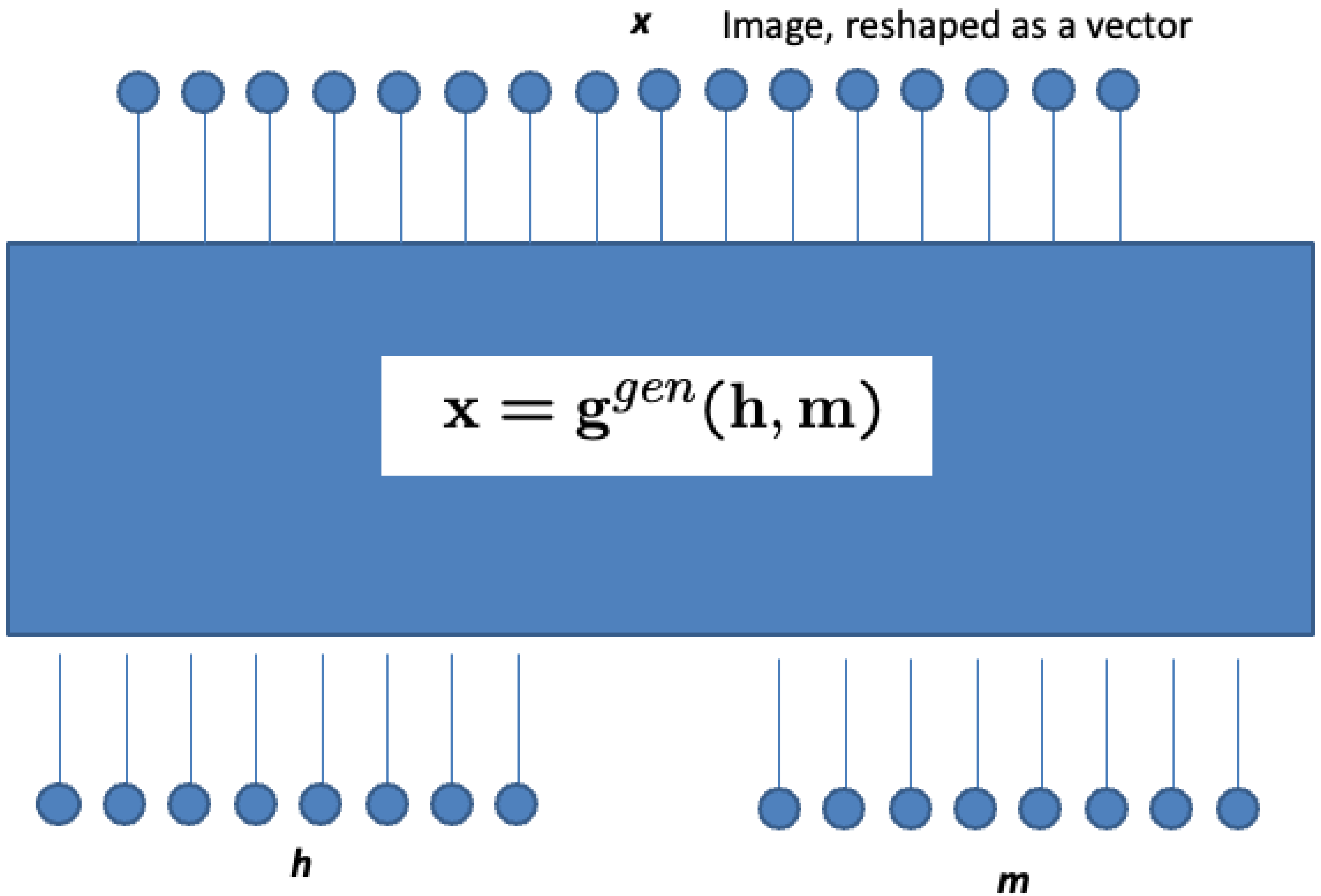


Requirement 2: Disentangled Representations



Even better: the dimensions have a meaning

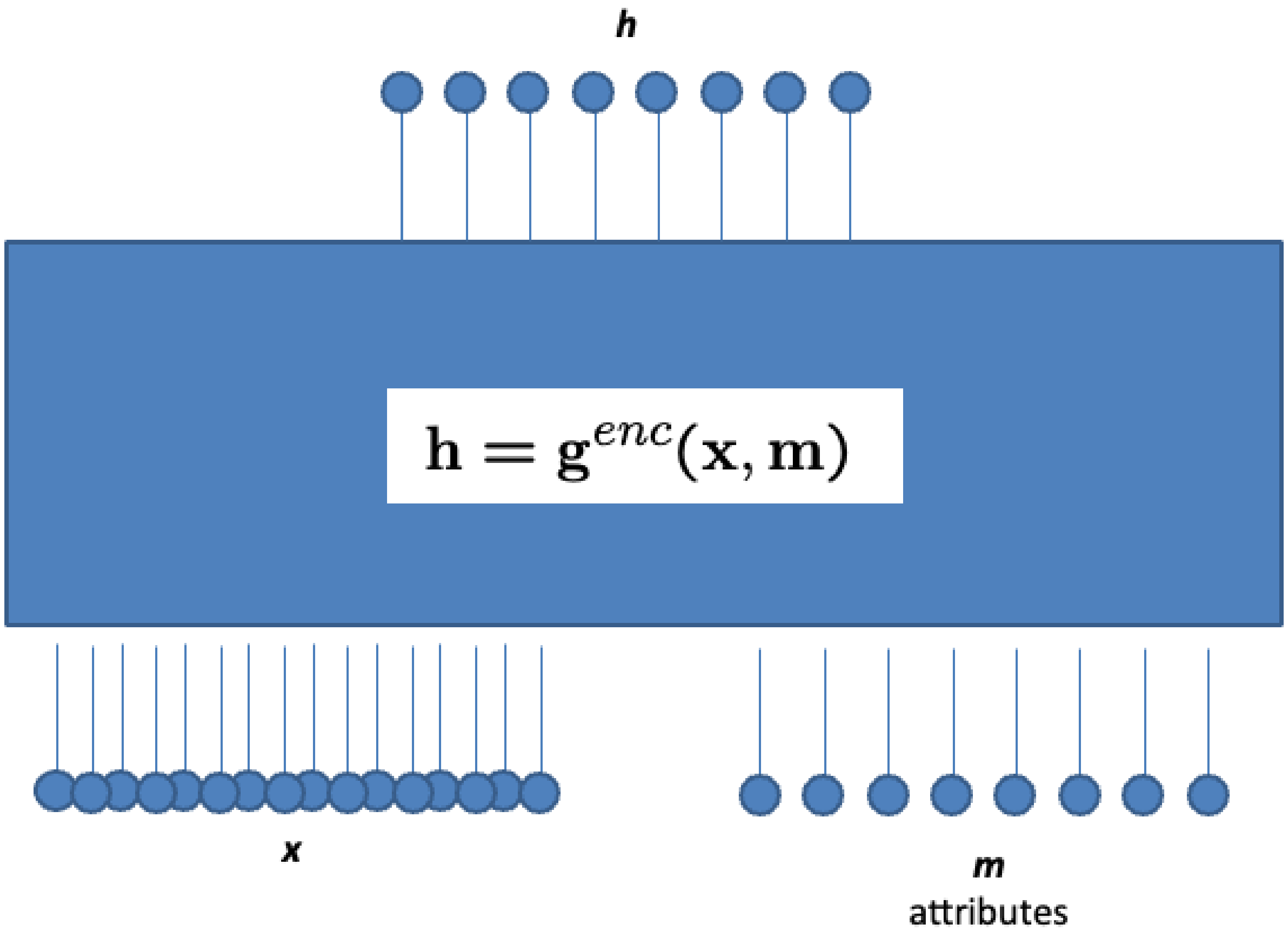
Requirement 3: Conditional



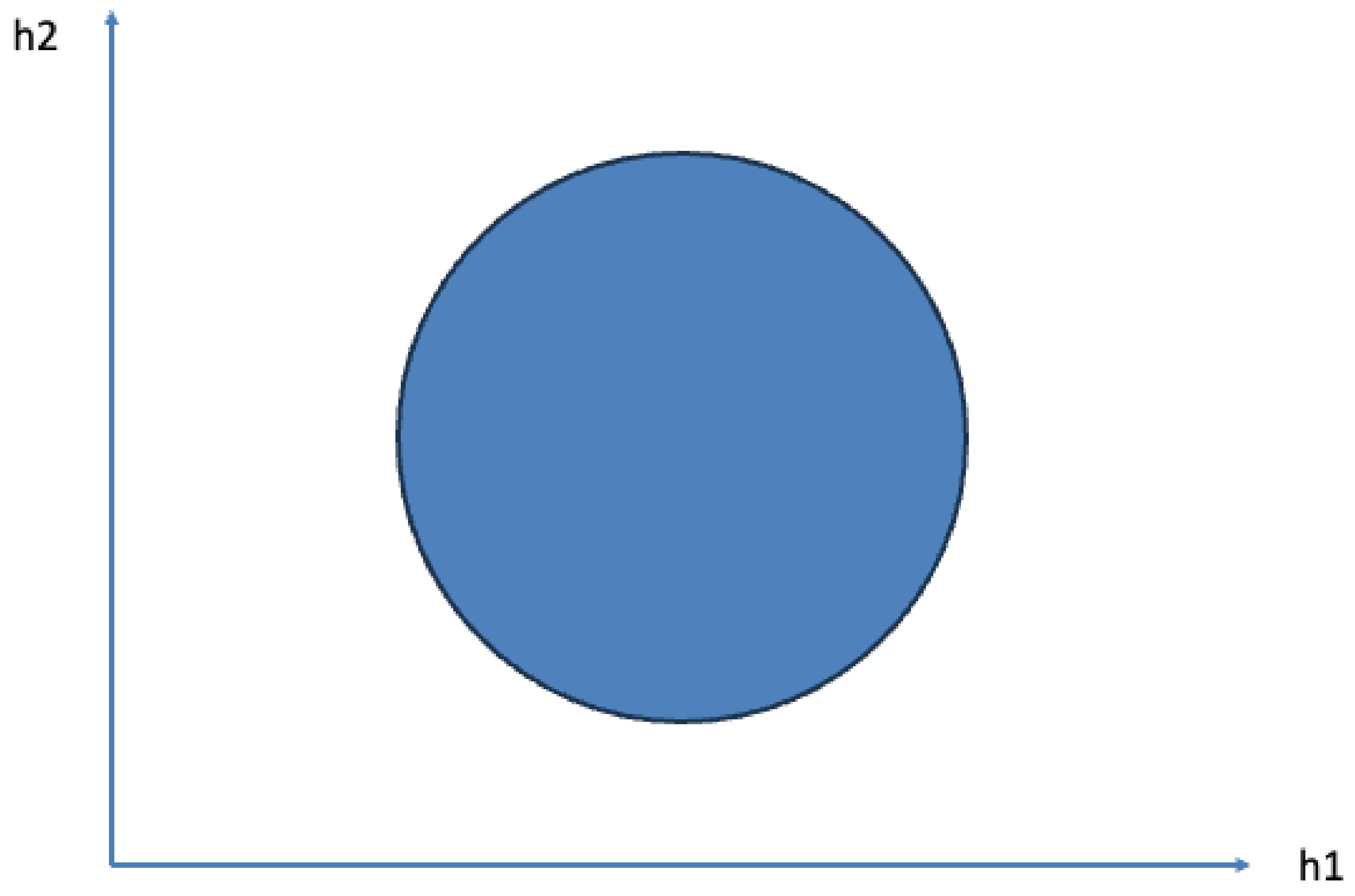
x Image, reshaped as a vector
$$\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h}, \mathbf{m})$$
h
m
desired attributes

- i.i.d. random inputs
- Disentangled: Each input might have an interpretation

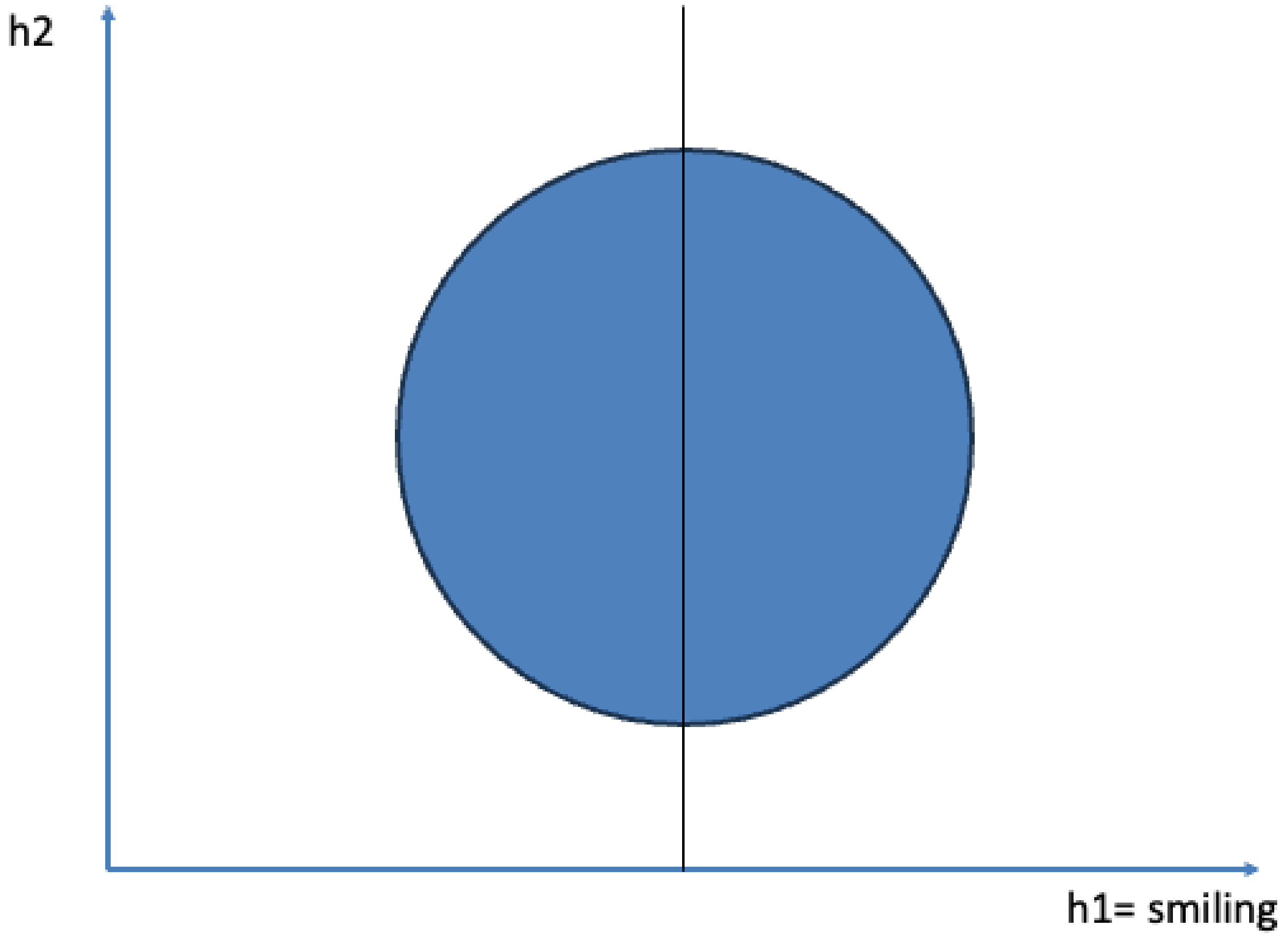
Conditional Encoder



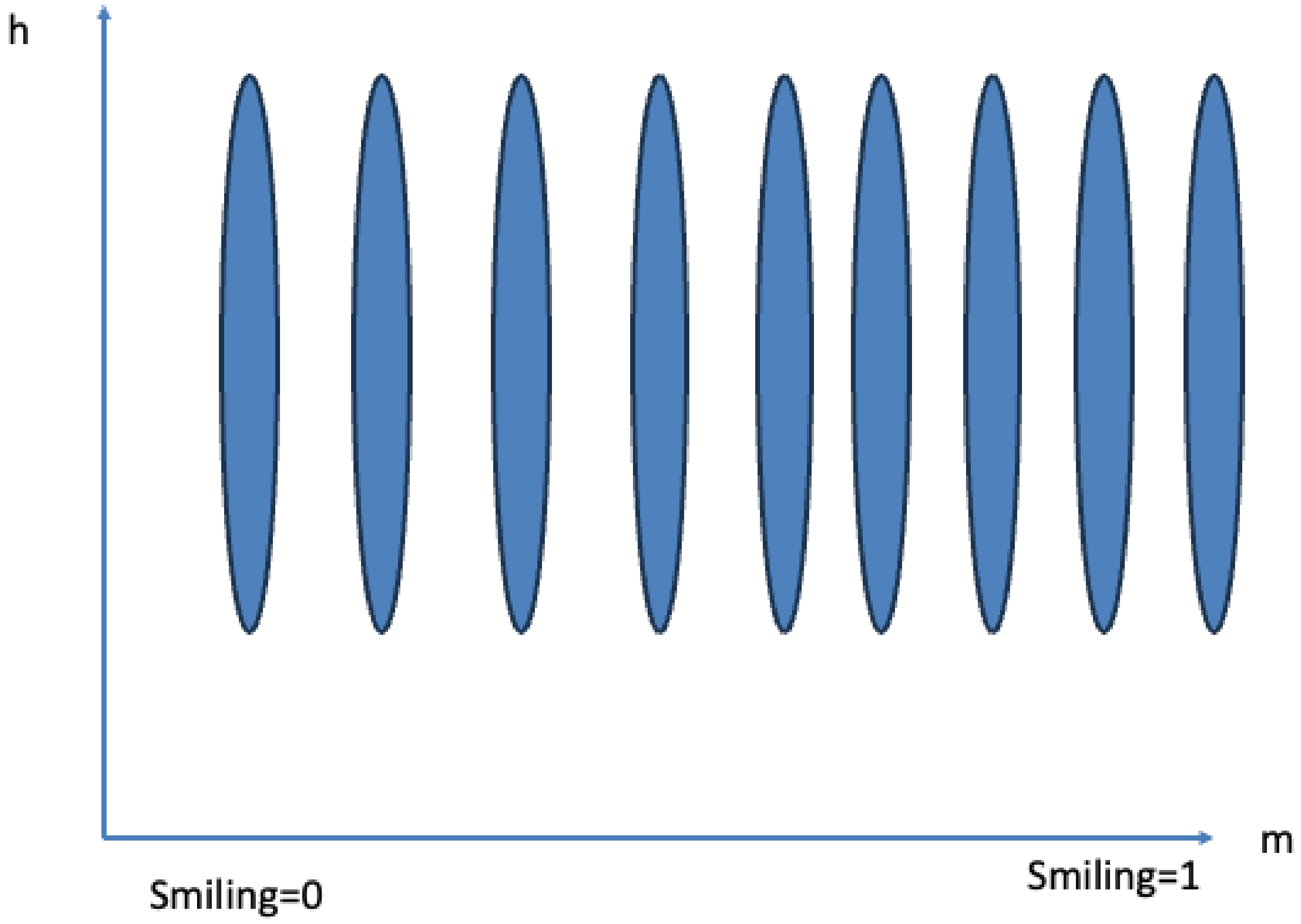
Mutli-variate Gaussian in latent space



Mutli-variate Gaussian in latent space with disentangled dimensions



Conditionned on attribute "smiling", we get independent Gaussians



Variational Autoencoder

Learning a Decoder (Generator)

- If I knew for each data point $\mathbf{h}$ and $\mathbf{x}$, I could learn a generator $\mathbf{g}^{gen}(\cdot)$ simply by supervised training
- Unfortunately, $\mathbf{h}$ is unknown
- Again, we assume that $\mathbf{h}$ comes from a Gaussian distribution with unit variance $P(\mathbf{h}) = \mathcal{N}(\mathbf{h}; 0, I)$ (each dimension is independently Gaussian distributed, with unit variance)
- Goal: **any** sample $\mathbf{h}$ from this Gaussian distribution should generate a valid **x** (this was not enforced in the normal autoencoder)
- This is the idea behind the Variational Autoencoder (VAE)

EM and Variational Learning

- Assume a training pattern $\mathbf{x}$ is given
- In EM learning, we need to calculate first $P(\mathbf{h}|\mathbf{x})$ (E-step) and based on this estimate we adapt the generator (M-step)
- Essentially, we can sample from $P(\mathbf{h}|\mathbf{x})$ and treat these samples as a true latent states
- The problem is that the E-step is intractable, since $P(\mathbf{h}|\mathbf{x})$ is complex and we cannot easily sample from it
- In the variational autoencoder (VAE), the variational approximation provides an approximate E-step
- $P(\mathbf{h}|\mathbf{x})$ is approximated by a Gaussian distribution; the mean (M_{hidd}-dimensional) and the dimension-specific variances of that Gaussian (M_{hidd} parameters) are predicted by the encoder

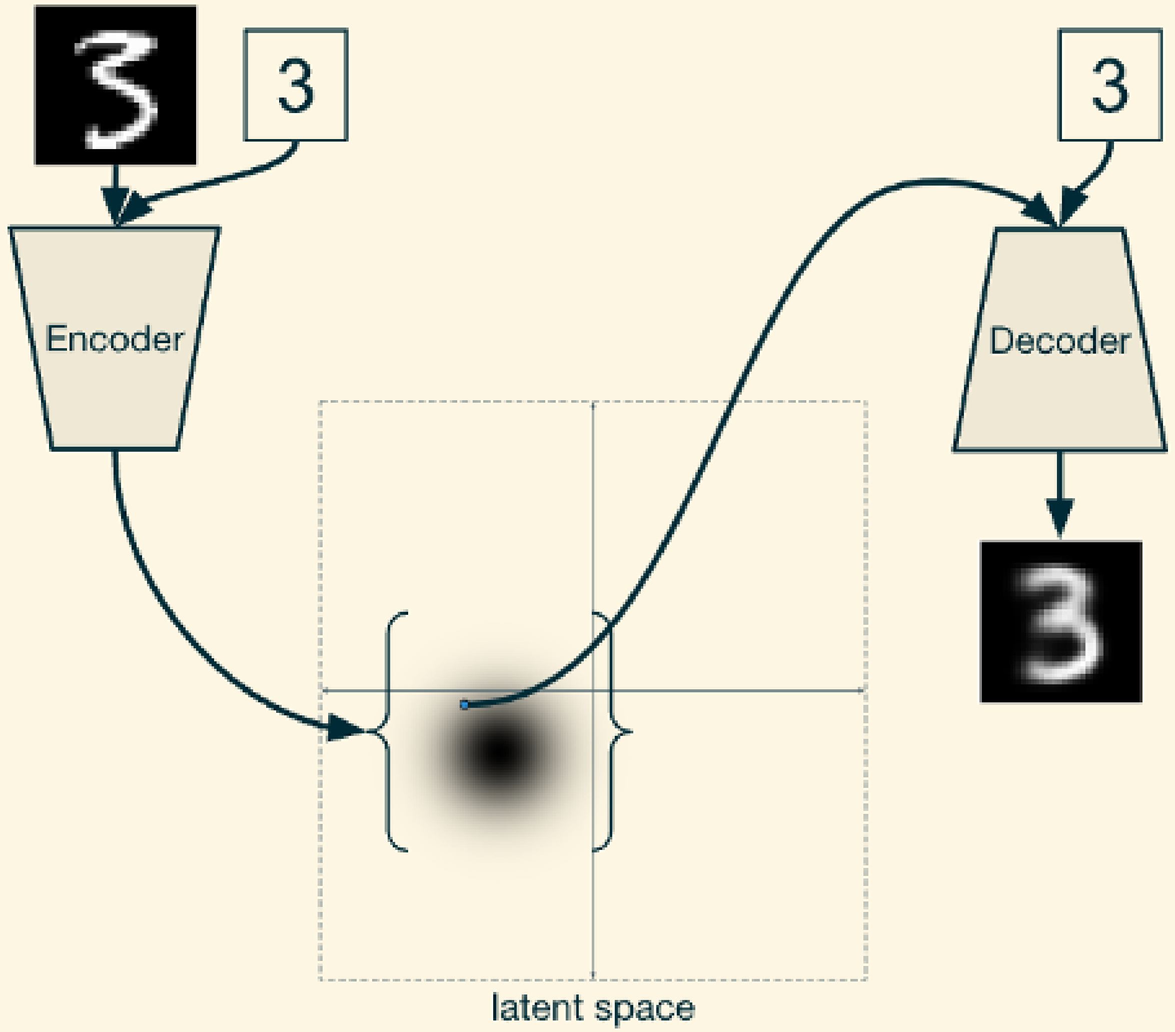
Generating Data from the VAE

- After training, we generate a new $\mathbf{x}$ by first generating a sample $\mathbf{h}_s$ from $\mathcal{N}(0, I)$; then

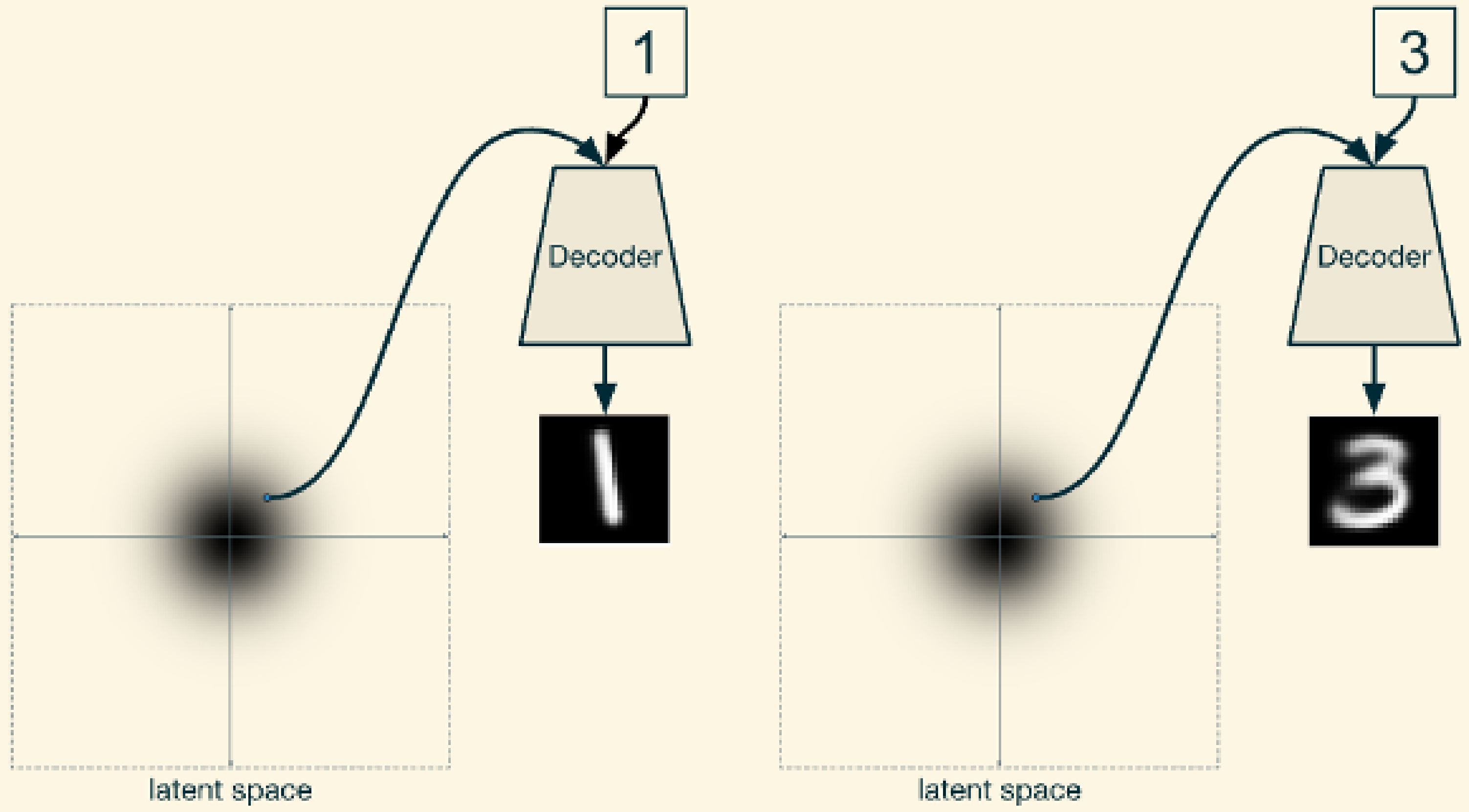
$$\mathbf{x} = \mathbf{g}^{gen}(\mathbf{h}_s)$$

Conditional Variational Autoencoders

- Conditional Variational Autoencoders
- $\mathbf{m}$ are inputs to both the generator and the encoder



A conditional variational autoencoder



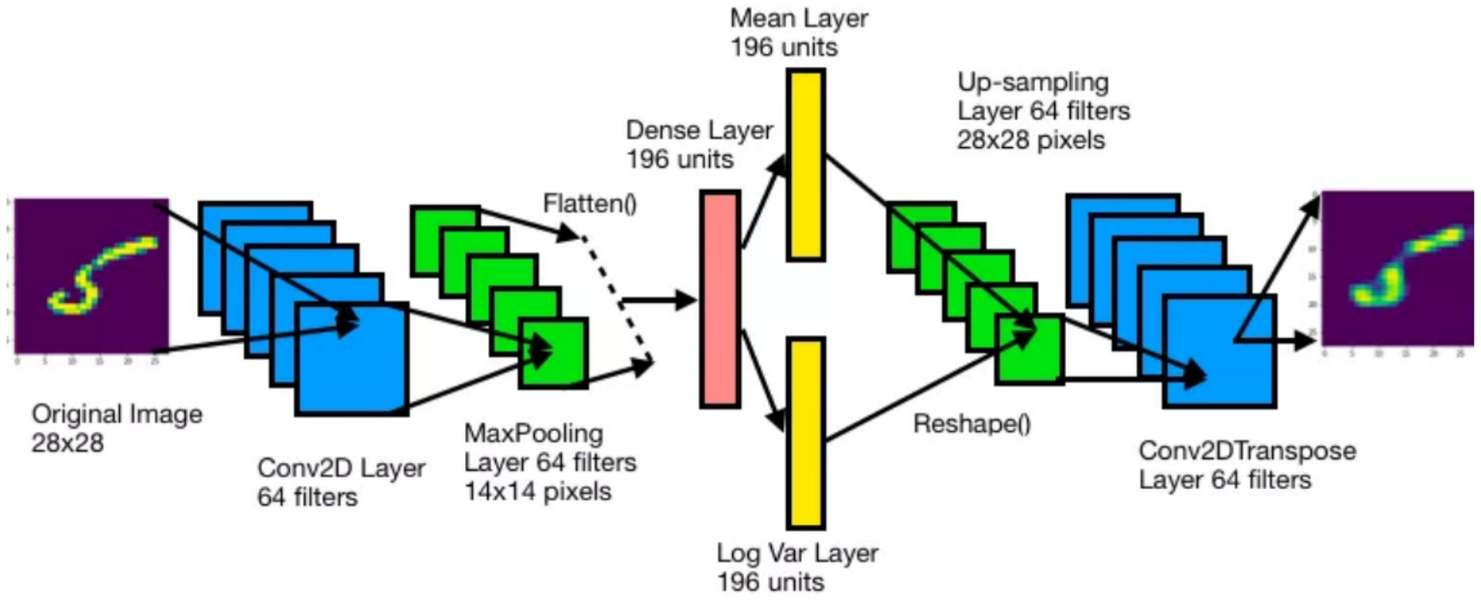
A variational autoencoder generating images according to given labels

The VAE is not Perfect

- For constrained, well-structured datasets (e.g., faces): 80-95% of images are valid/useful.
- For complex datasets (e.g., diverse natural scenes): 50-80% of images may be valid/useful.
- With improper training or sampling outside the prior: The percentage can drop significantly.

Convolutional Variational Autoencoders

- The convolutional variational autoencoder uses convolutional layers



Convolutional Variational Autoencoder

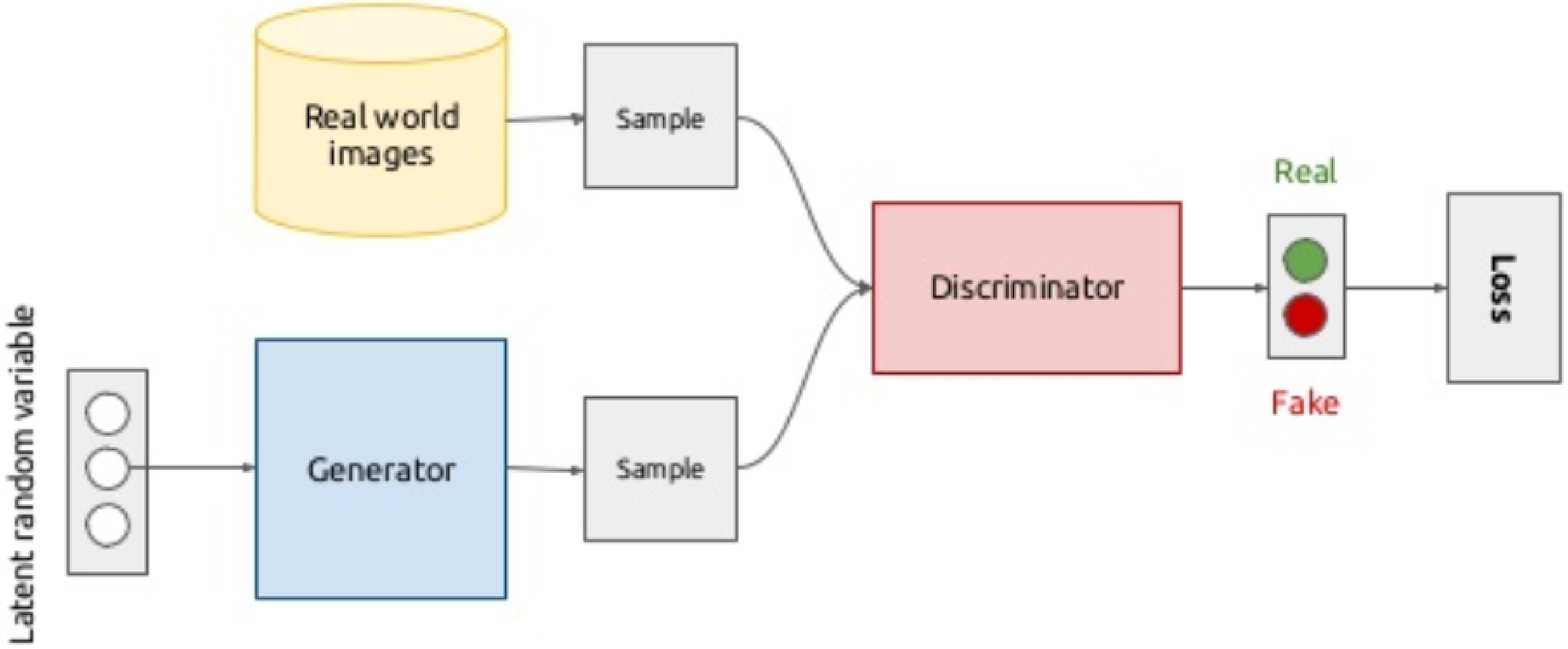
Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs)

- Can we train a decoder (generator) without without an encoder?
- Let's assume we have a larger number of **generators** available; let's assume that each generator generates a data set; which one is the best generator?
- The best generator might be the one where a **discriminator** (i.e., a binary neural network classifier) trained to separate training data and the data from a particular generator, cannot separate the two classes; if this is the case, one might say that

$$P(\mathbf{x}) \approx P^{gen}(\mathbf{x})$$

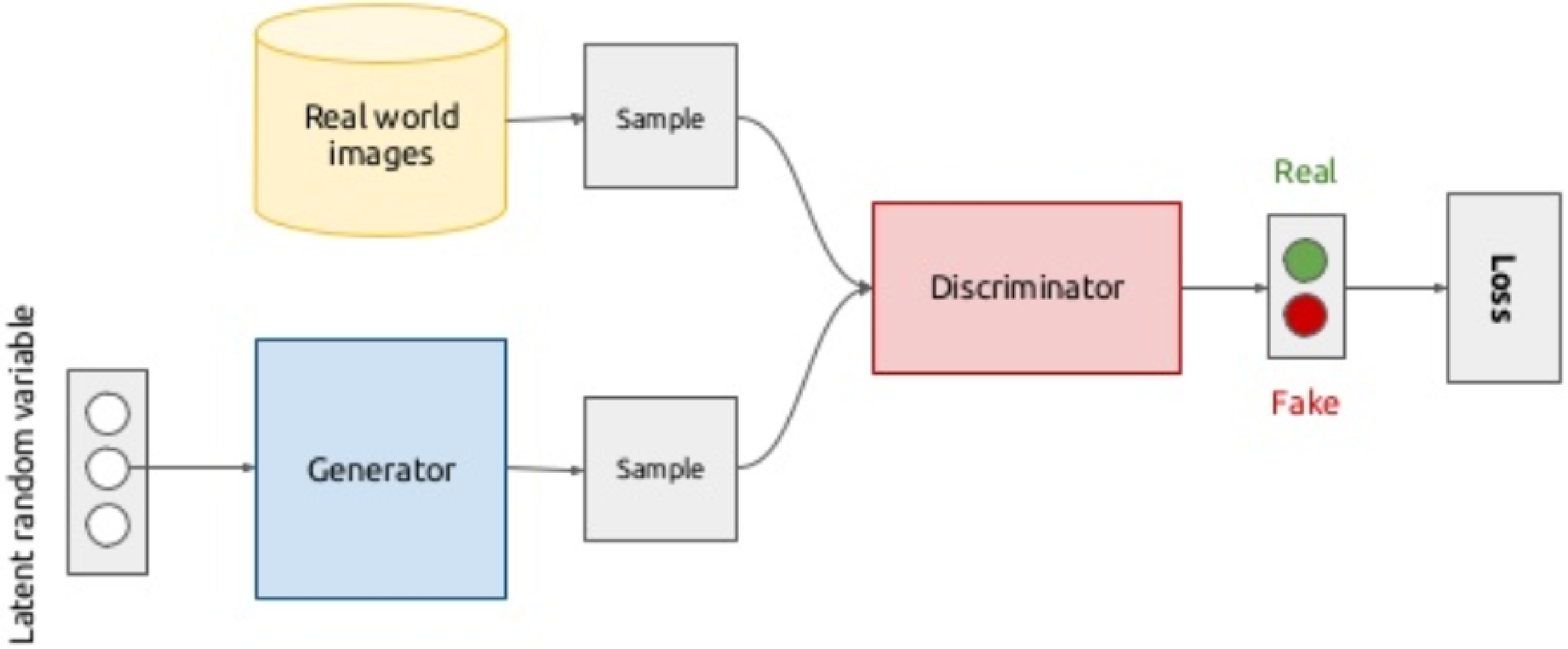
- In GAN models, there is only one generator and one discriminator and both are trained jointly


Real world images
Sample
Latent random variable
Generator
Sample
Discriminator
Real
Fake
Loss

Cost Function

- **Discriminator:** Given a set of training data and a set of generated data: the weights $\mathbf{w}$ in the discriminator are updated to **maximize** the negative cross entropy cost function (i.e., the log-likelihood); the targets for the training data are 1 and for the generated data 0 (this is the same as minimizing the cross entropy, i.e. the discriminator is trained to be the best classifier possible)

- **Generator:** With a given discriminator and a set of latent samples: update the weights $\mathbf{v}$ in the generator, such that the generated data get closer to the classification decision boundary: the generator is trained to **minimize** the negative cross entropy cost function, where backpropagation is performed through the discriminator (this is the same as maximizing the cross entropy)


Real world images
Sample
Latent random variable
Generator
Sample
Discriminator
Real
Fake
Loss

Parameter Learning

- Optimal parameters are

$$(\mathbf{w}, \mathbf{v}) = \arg\max_{\mathbf{v}} \arg\min_{\mathbf{w}} \text{cost}(\mathbf{w}, \mathbf{v})$$

where (assuming the numbers of real images and generated images are the same)

$$\text{cost}(\mathbf{w}, \mathbf{v}) = -\sum_{i:\mathbf{x}_i \in \text{train}} \log g^{\mathit{dis}}(\mathbf{x}_i, \mathbf{w}) - \sum_{i':\mathbf{x}_{i'} \in \text{gen}} \log[1 - g^{\mathit{dis}}(g^{\mathit{gen}}(\mathbf{h}_{i'}, \mathbf{v}), \mathbf{w})]$$

- The left sum says: all actual images should be classified by the discriminator with label 1

- The right sum says: all generated images should be classified by the discriminator with label 0

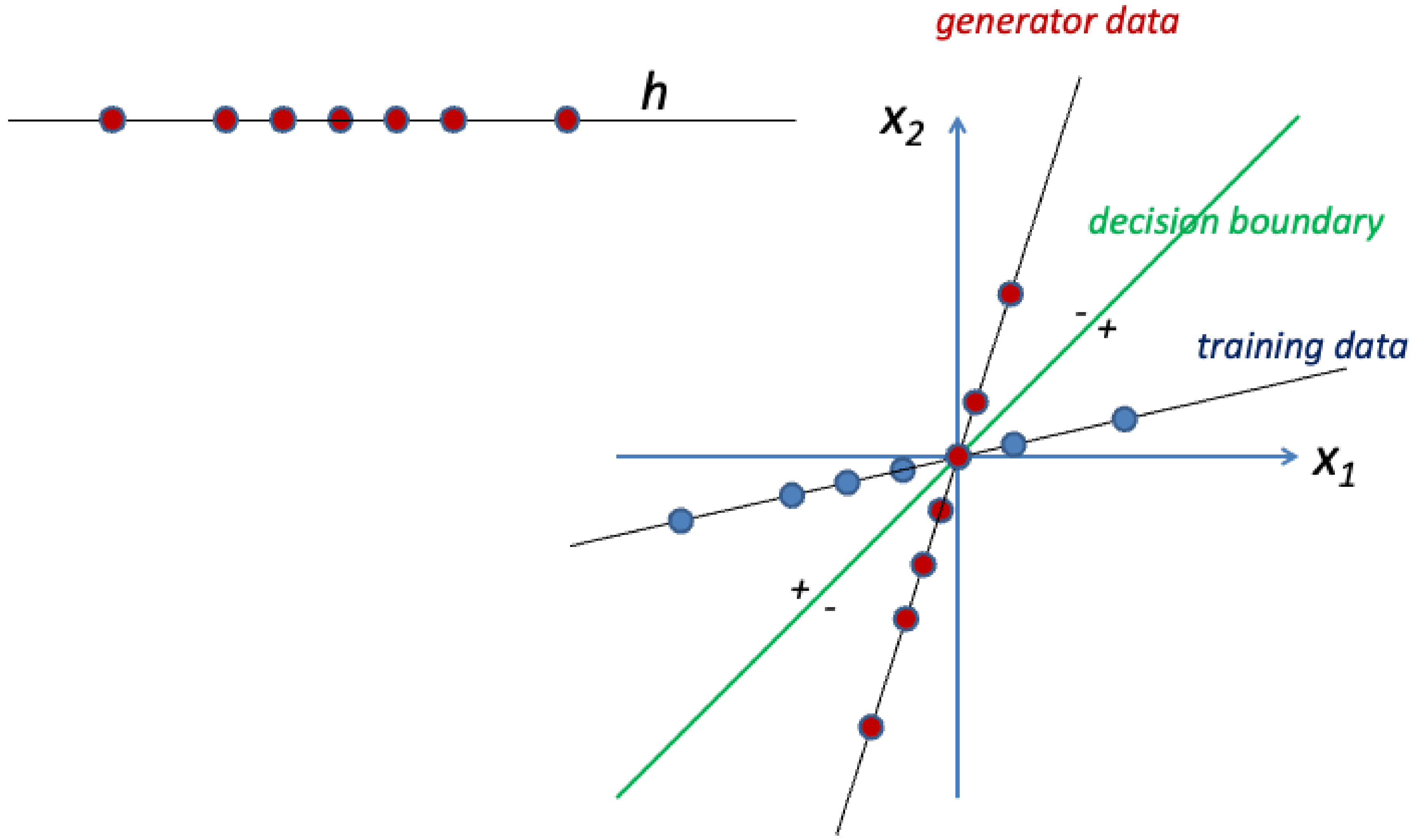
- $g^{\mathit{gen}}(\mathbf{h}_{i'}, \mathbf{v})$ is a generated image with a random $\mathbf{h}_{i'}$

Minimax

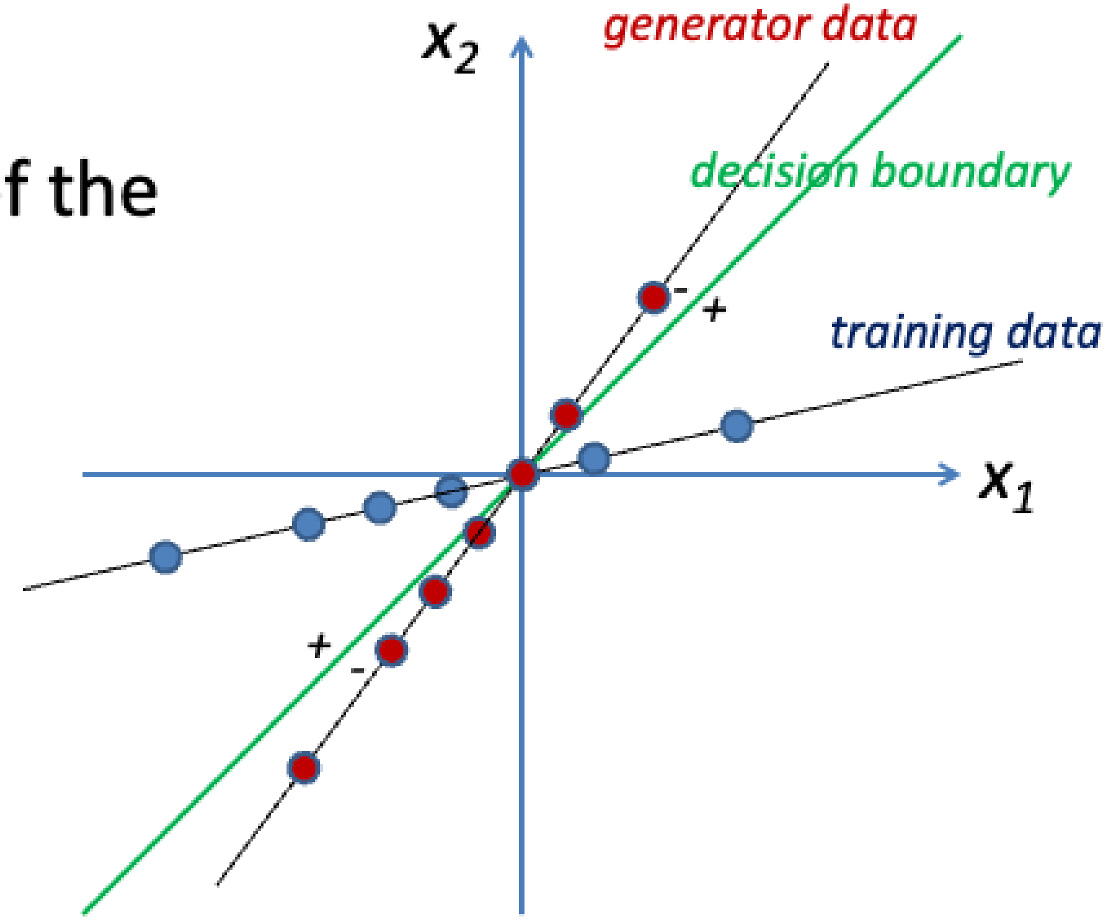
- This can be related to game theory: The solution for a zero-sum game is called a minimax solution, where the goal is to minimize the maximum cost for the player, here the generator. The **generator** wants to find the lowest cost, without knowing the actions of the **discriminator** (in two-player zero-sum games, the minimax solution is the same as the Nash equilibrium)

Illustration

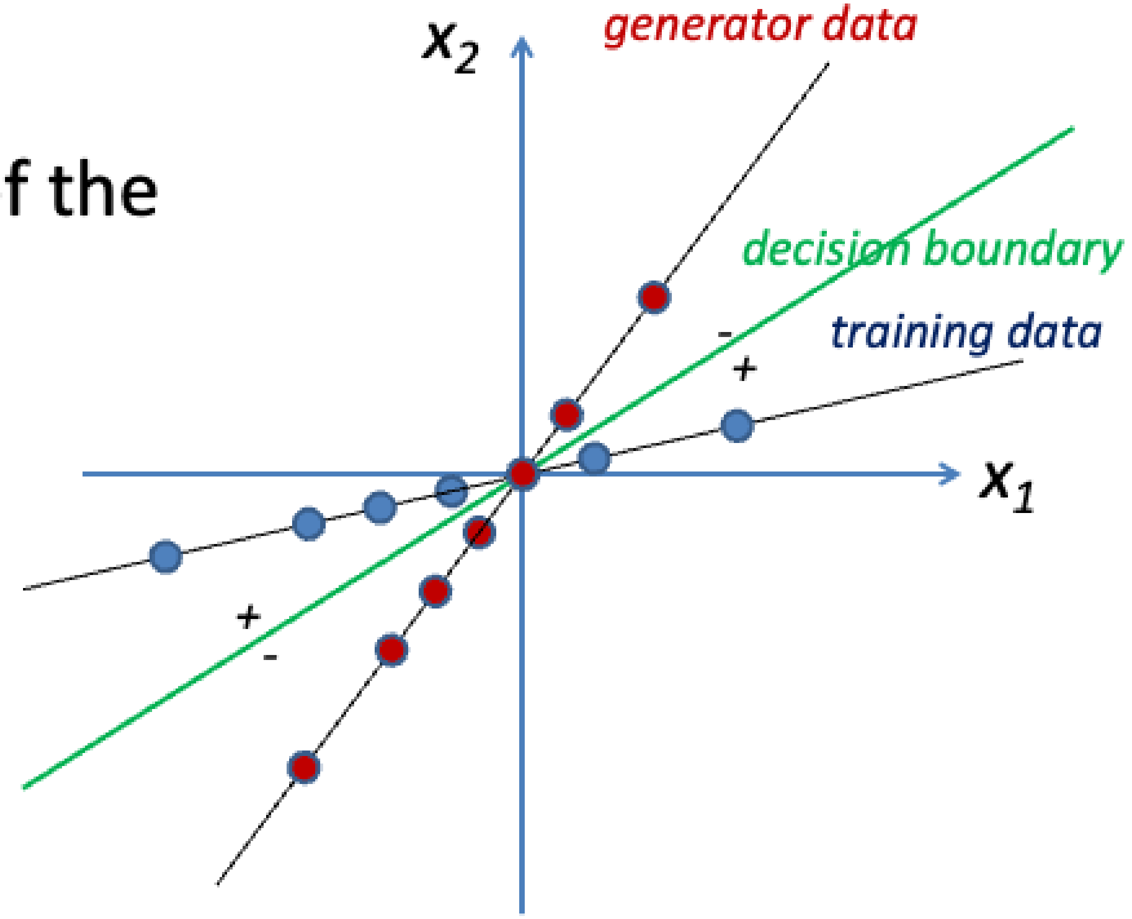
- Consider the following figure; h is one-dimensional Gaussian distributed: $P(h) = \mathcal{N}(h; 0, 1)$, $H = 1$
- The generator is $\mathbf{x} = h\mathbf{v}$, where $M = 2$; the data points are on a 1-D manifold in 2-D space; here: $v_1 = 0.2$, $v_2 = 0.98$
- The training data are generated similarly, but with $\mathbf{x} = h\mathbf{w}$ and $w_1 = 0.98$, $w_2 = 0.2$
- The discriminator is $y = \text{sig}(|x_1|w_1 + |x_2|w_2)$, with $w_1 = 0.71$, $w_2 = -0.71$
- After updating the generator, we might get $v_1 = 0.39$, $v_2 = 0.92$
- After updating the discriminator, we might get $w_1 = 0.67$, $w_2 = -0.74$


generator data
h
x_2
decision boundary
- +
training data
x_1
+ -

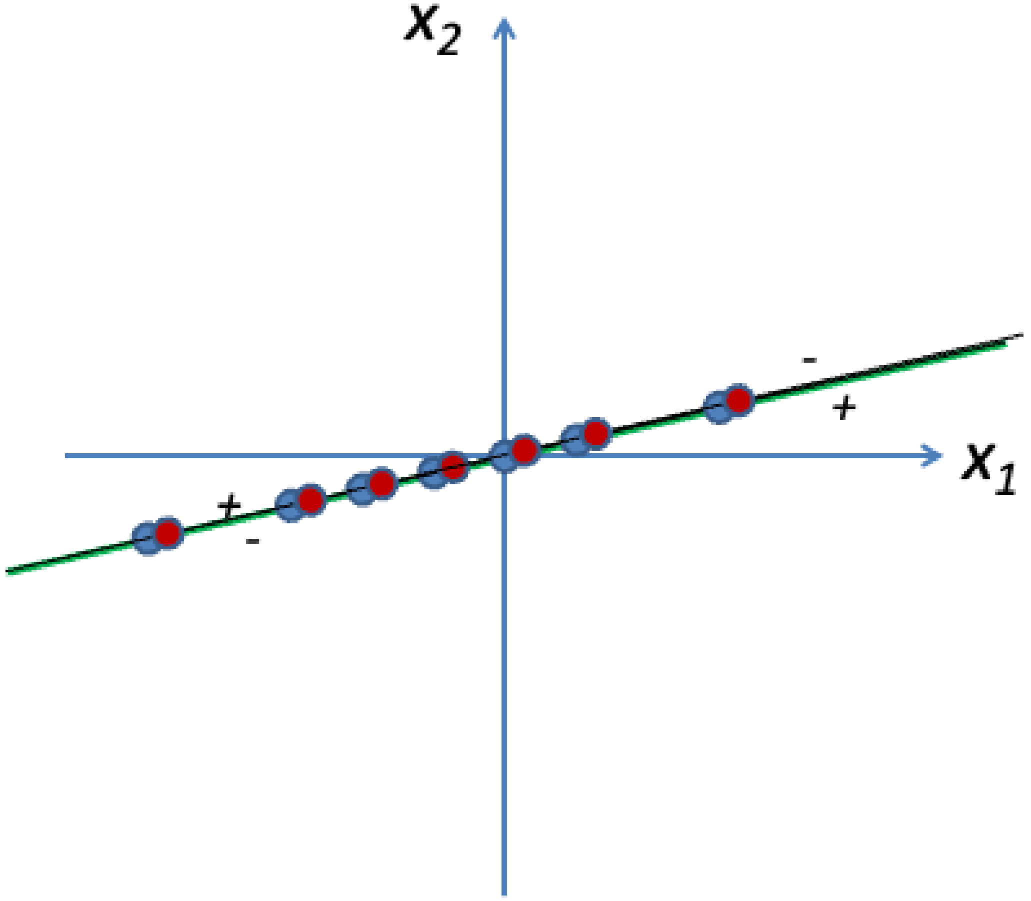
After update of the generator



After update of the discriminator



Convergence



Modifying the Cost Function

- There have been modifications to the cost function to make learning faster and more stable

$$\text{cost}(\mathbf{w},\mathbf{v}) = -\sum_{i:\mathbf{x}_i\in\text{train}} \log g^{dis}(\mathbf{x}_i,\mathbf{w}) + \sum_{i':\mathbf{x}_{i'}\in\text{gen}} \log[g^{dis}(g^{gen}(\mathbf{h}_{i'},\mathbf{v}),\mathbf{w})]$$

- Wasserstein GAN

$$\text{cost}(\mathbf{w},\mathbf{v}) = -\sum_{i:\mathbf{x}_i\in\text{train}} \left[\log g^{dis}(\mathbf{x}_i,\mathbf{w}) - \lambda\left(\|\nabla_{\mathbf{x}_i} g^{dis}(\mathbf{x}_i,\mathbf{w})\|^2 - 1\right)^2\right]$$
$$+ \sum_{i':\mathbf{x}_{i'}\in\text{gen}} \log[g^{dis}(g^{gen}(\mathbf{h}_{i'},\mathbf{v}),\mathbf{w})]$$

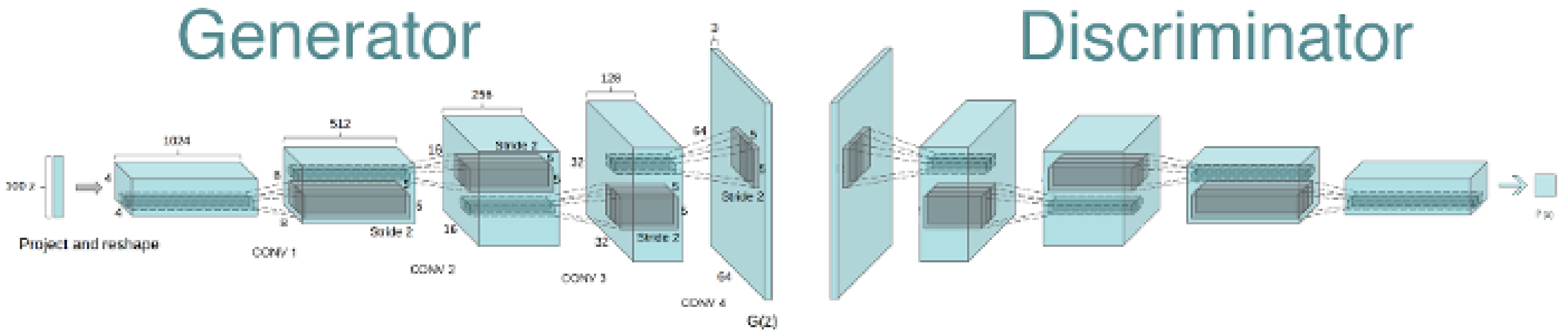
Applications

- For discriminant machine learning: Outputs of the convolutional layers of the discriminator can be used as a feature extractor, with simple linear models fitted on top of these features using a modest quantity of (image-label) pairs
- For discriminant machine learning: When labelled training data is in limited supply, adversarial training may also be used to synthesize more training samples

DCGAN

- If the data consists of images, the discriminator is a binary image classifier and the generator needs to generate images
- Deep Convolutional GAN (DCGAN): the generator and the discriminator contain convolutional layers and transposed convolution layers
- The transposed convolution layer is sometimes (incorrectly) called deconvolution layer (a deconvolution is really something different)
- The generation of sharp, photo realistic images with sharp edges and smooth regions is nontrivial!
- Radford et al. (shown below). $H = 100$; samples drawn from a uniform distribution (we refer to these as a code, or latent variables) and outputs an image (in this case $64 \times 64 \times 3$ images (3 RGB channels)

DCGAN


Generator
Discriminator
100 z
Project and reshape
1024
4
4
512
8
8
Stride 2
5
5
CONV 1
256
16
16
Stride 2
5
5
CONV 2
128
32
32
Stride 2
5
5
CONV 3
64
64
3
Stride 2
5
5
CONV 4
G(z)
P(x)

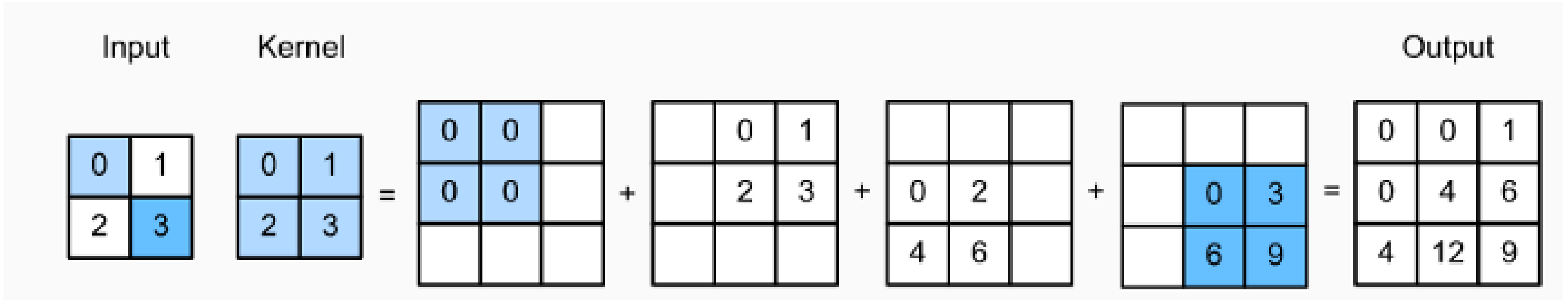
Convolution and Transposed Convolution

- With linear neurons, and the first hidden layer is

$$\mathbf{h} = \mathbf{W}^T \mathbf{x}$$

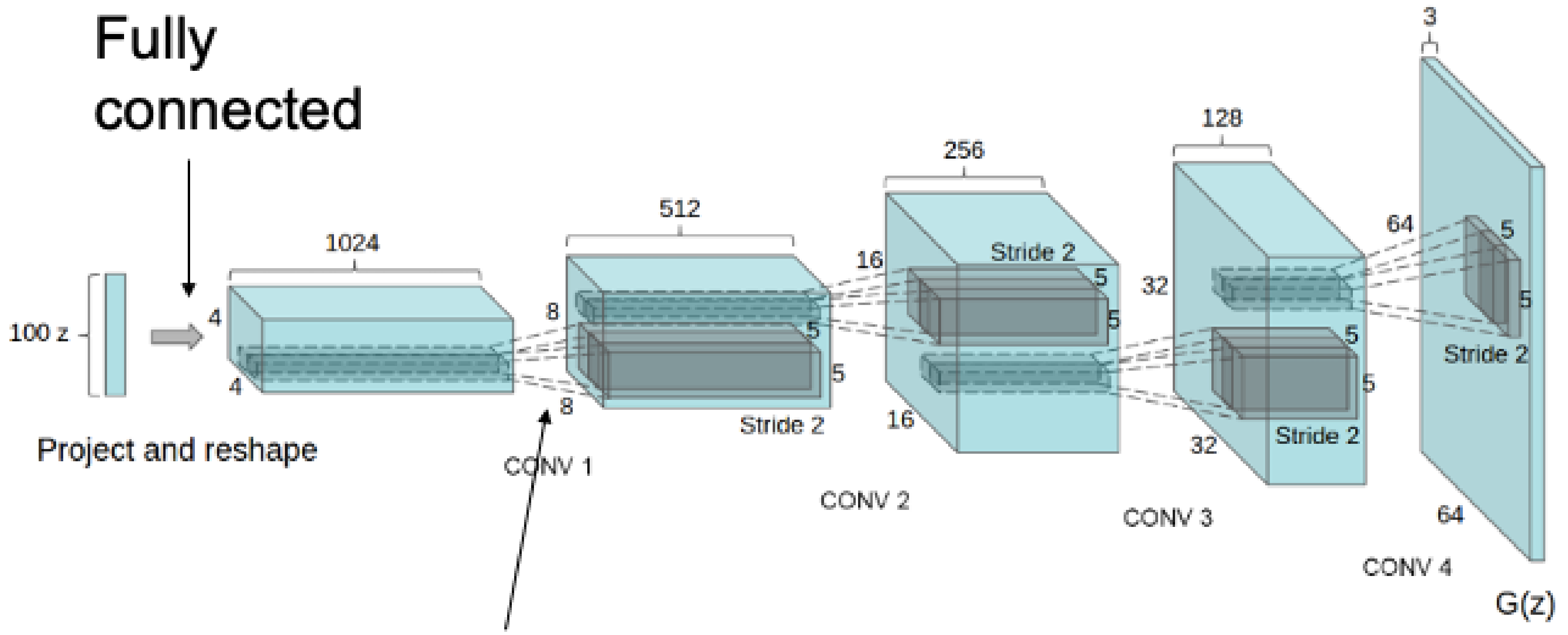
- If $\mathbf{W}$ is orthonormal, then $\mathbf{x} = (\mathbf{W})_{:,k}$ will only activate hidden unit k
- Similarly, if we only activate hidden unit k, and propagate towards the inputs, we will get again the same pattern, $\mathbf{x} = (\mathbf{W})_{:,k}$
- In a real convolutional NN, $\mathbf{W}$ is defined by the kernels and is not orthonormal; propagating back is known as a **transposed convolution** (which is also a convolution)
- Transposed convolution generates the characteristic input pattern of the hidden neurons, i.e. for hidden unit k, $\mathbf{x} = (\mathbf{W})_{:,k}$
- If $\mathbf{W}$ is not orthonormal, the transposed convolution is not a deconvolution; the latter would implement the inverse of the convolution and would require a very different connection matrix

Transposed Convolution ("Deconvolution")



DCGAN: Generator

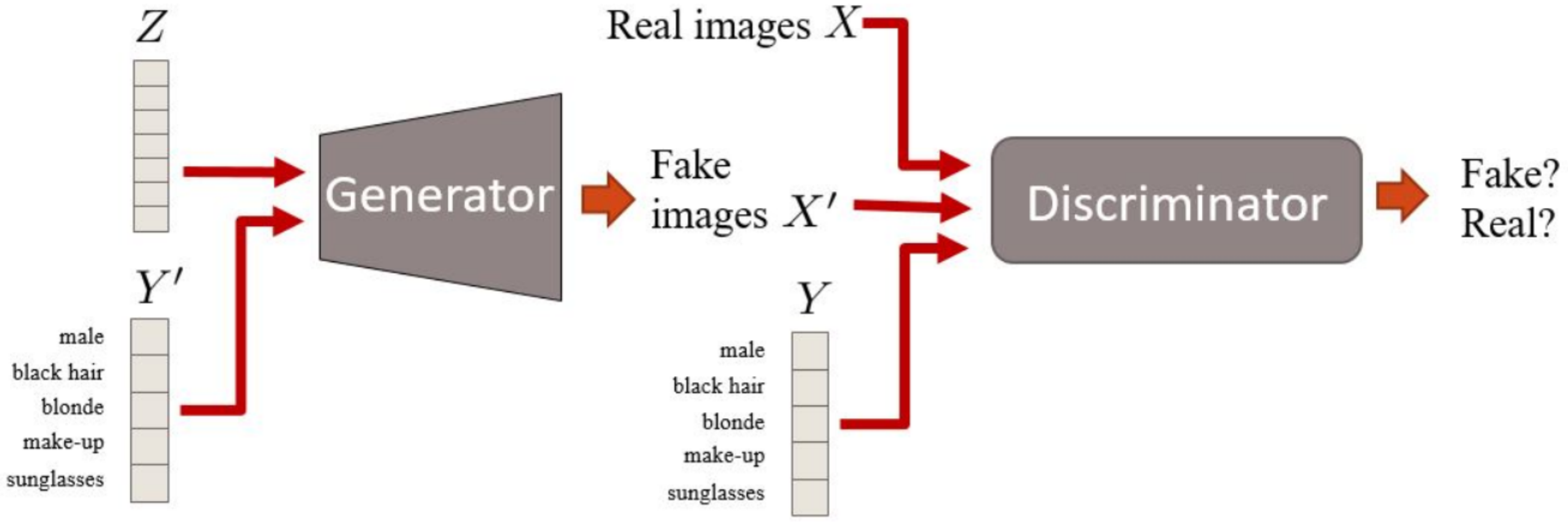
Fully connected



(the general scheme is :
transposed convolution→ batch normal→leaky ReLU)

cGAN

- Consider that class/attribute labels are available; in a normal GAN, one would ignore them; another extreme approach would be to train a different GAN model for each class
- Conditional GAN (cGAN): An additional input to the generator and the discriminator is the class/attribute label



Overview of a conditional GAN with face attributes information.

cGAN Applications

- The attributes can be quite rich, e.g., images, sketches of images
- cGANs: GAN architecture to **synthesize images from text descriptions**, which one might describe as reverse captioning. For example, given a text caption of a bird such as "white with some black on its head and wings and a long orange beak", the trained GAN can generate several plausible images that match the description
- cGANs not only allow us to synthesize novel samples with specific attributes, they also allow us to develop **tools for intuitively editing images** - for example editing the hair style of a person in an image, making them wear glasses or making them look younger
- cGANs are well suited for **translating an input image into an output image**, which is a recurring theme in computer graphics, image processing, and computer vision

Unpaired Image-to-Image Translation

- Example task: turn horses in images into zebras
- One could train a generator *Generator A2B* with horse images as inputs and the corresponding zebra images as output; this would not work, since we do not have matching zebra images
- But consider that we train a second generator *Generator B2A* which has zebra images as inputs and generates horse images
- Now we can train two autoencoders

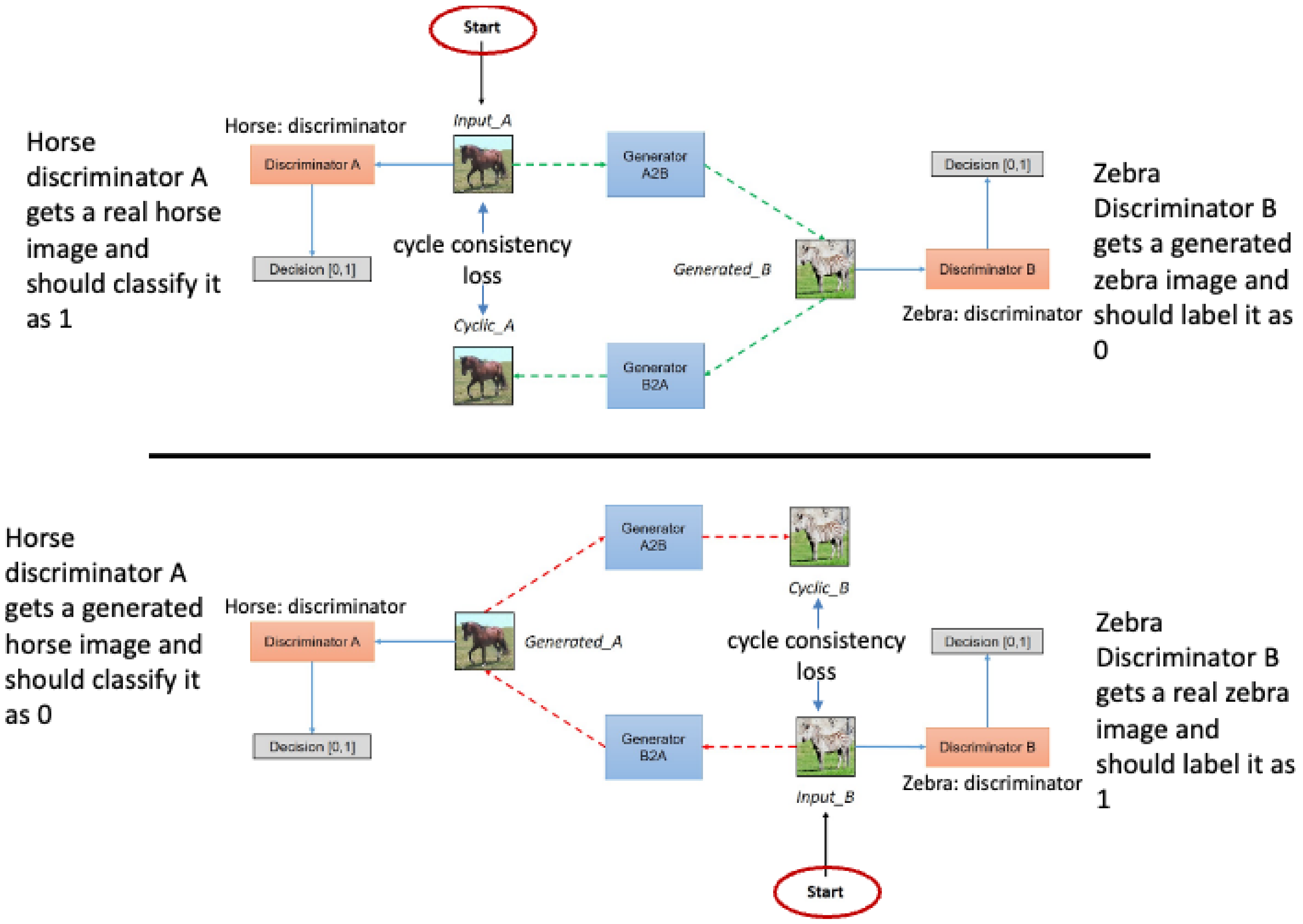
$$\hat{\mathbf{x}}_{horse} = g_{B2A}(g_{A2B}(\mathbf{x}_{horse}))$$

$$\hat{\mathbf{x}}_{zebra} = g_{A2B}(g_{B2A}(\mathbf{x}_{zebra}))$$

- These constraints are enforced using the *cycle consistency loss*, $\|\mathbf{x}_{horse} - \hat{\mathbf{x}}_{horse}\|^2$ and $\|\mathbf{x}_{zebra} - \hat{\mathbf{x}}_{zebra}\|^2$

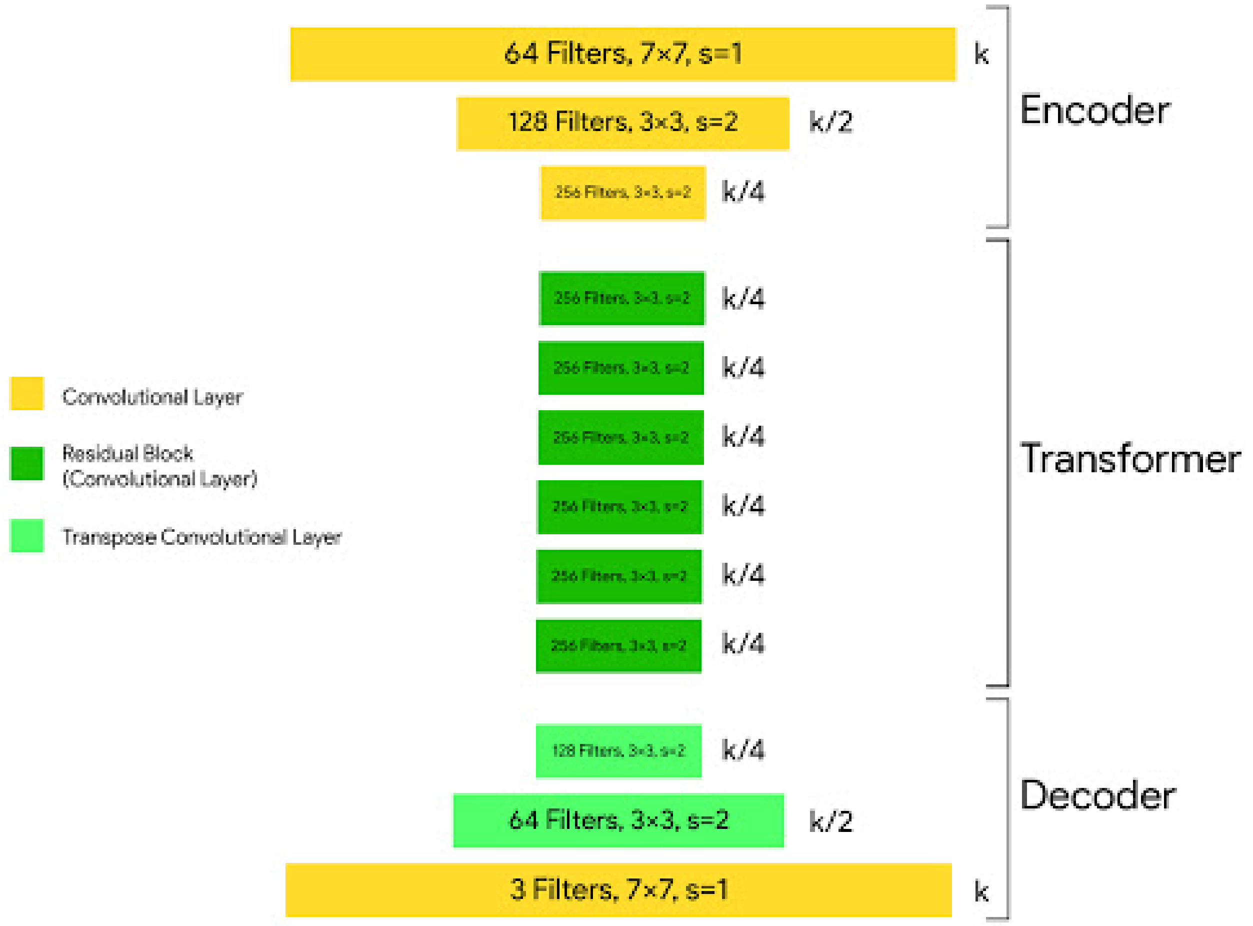
CycleGAN

- CycleGAN does exactly that
- CycleGAN adds two discriminators, trained with the *adversarial loss*
- *discriminator*$_A$ tries to discriminate real horses from generated horses
- *discriminator*$_B$ tries to discriminate real zebras from generated zebras
- If the generated horses and zebras are perfect, both fail to discriminate
- Both the *cycle consistency loss* and the *adversarial loss* are used in training
- Note that the random seeds here are the images!



Simplified view of CycleGAN architecture

Generator for CyleGAN



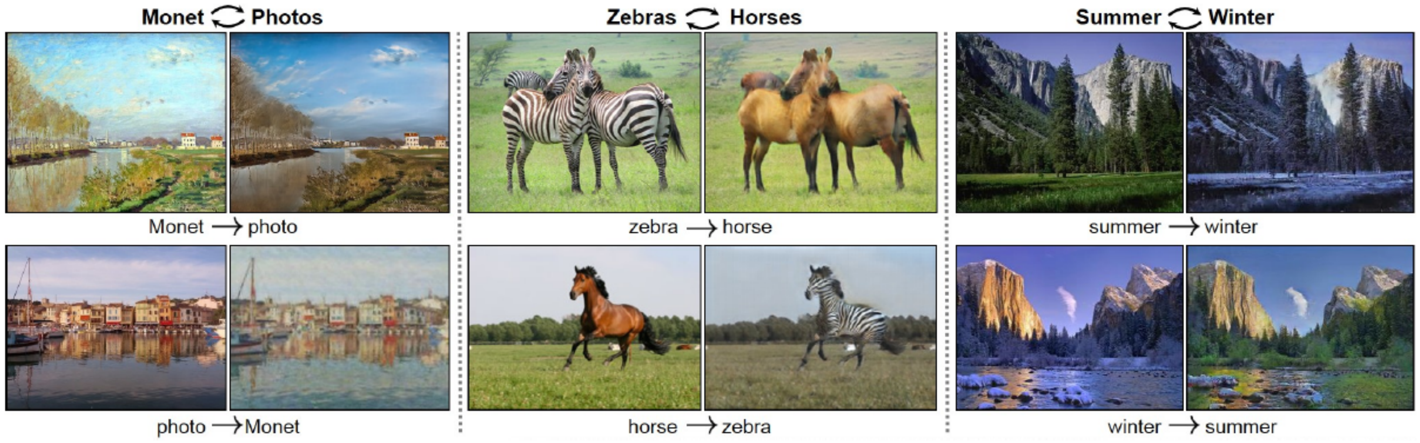
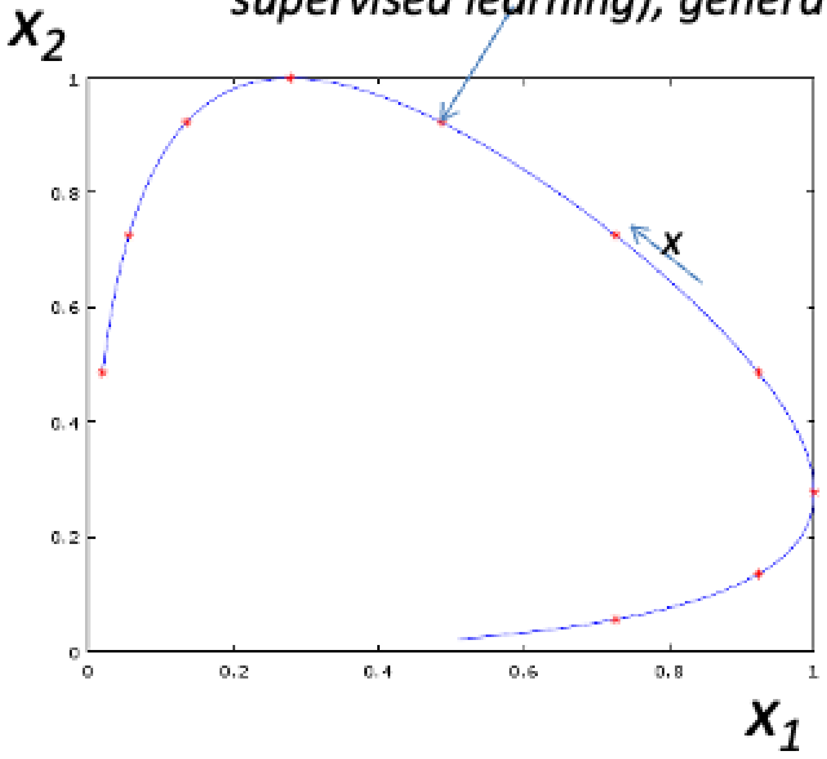


Fig. 8. CycleGAN model learns image to image translations between two unordered image collections. Shown here are the examples of bi-directional image mappings: Monet paintings to landscape photos, zebras to horses, and summer to winter photos in Yosemite park. Figure reproduced from [4].

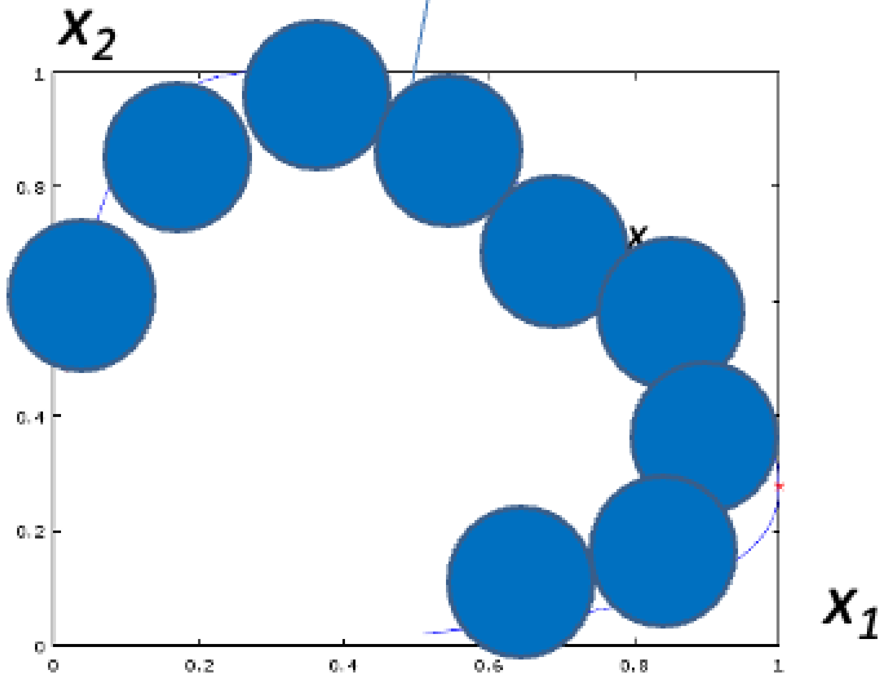
Why Not Use Classical Approaches?

- Classically, one would start with a probabilistic model $P(\mathbf{x}; \mathbf{w})$ and determine parameter values that provide a good fit (maximum likelihood)
- Examples for continuous data: Gaussian distribution, mixture of Gaussian distributions
- These models permit the specification of the probability density for a new data point and one can sample from these distributions (typically by transforming samples for a normal or uniform distribution; this would be the generator here)
- These approaches typically work well for low dimensional distributions, but not for image distributions with 256×256 pixels and where data is essentially on a manifold

AE and GANs can learn the manifold distribution (see semi-supervised learning); generated face images are sharp and realistic



Classical models are too rough and cannot model the manifold very well: generated face images are nonrealistic and blurry



Related Approaches and Applications

- The GAN generator generates data $\mathbf{x}$ but we cannot easily evaluate $P(\mathbf{x})$
- In many applications it is possible to generate data but one cannot generate a likelihood function (likelihood free methods)
- Moment matching is one approach to evaluate the quality of the simulation
- Optimization: in physics and other fields it is sometimes easy to evaluate the cost (e.g., energy) of a solution and the problem is to find good proposal solutions $\mathbf{x}$ with a low $\mathrm{cost}(\mathbf{x})$ (combinatorical optimization)