

Deep learning

Deep Generative Models

Hamid Beigy

Sharif University of Technology

December 31, 2022





1. Introduction
2. Deep generative models
3. Autoencoder models
4. Autoregressive models
5. Generative Adversarial Networks
6. Variational Autoencoder models
7. Normalizing Flow Models
8. Evaluating deep generative models
9. Summary
10. Reading

Introduction



1. In **supervised** setting, we have a dataset $S = \{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$.
2. **Discriminative models** estimate the conditional distribution $P(y|x)$.
 - ▶ Linear regression, logistic regression, generalized linear models
 - ▶ Standard Neural Networks, CNN, RNN...
 - ▶ Decision trees, boosting, random forests, kernel methods, KNN, ...
3. **Generative models** estimate the joint distribution $P(x, y)$.
 - ▶ Naive Bayes
 - ▶ Linear/quadratic discriminant analysis
4. **Generating new data** requires to model the joint distribution $P(x, y)$.

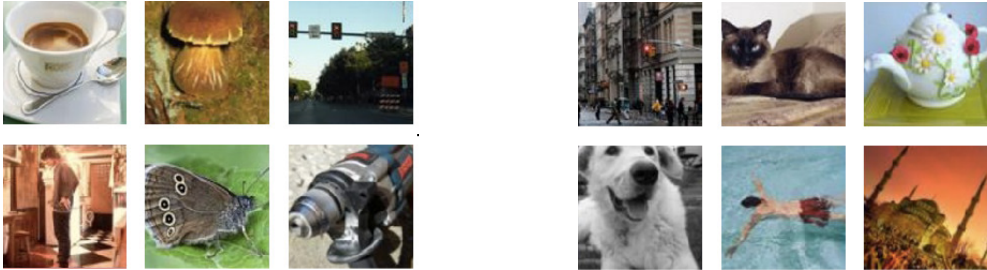


1. In **unsupervised** setting, we have a dataset $S = \{x_1, x_2, \dots, x_m\}$.
2. We have **no target output**, thus nothing to predict nor discriminate.
3. In **unsupervised** setting, we have different goals:
 - ▶ **Descriptive analysis**: detect structure, correlations in the data set using descriptive/graphical tools or using more involved methods (PCA for example)
 - ▶ **Clustering**: create "homogeneous" groups of observations (usually spending 90% of the allocated time to properly define "homogeneous")
 - ▶ **Estimating the distribution of observations**: detect suspect data/behaviour, detect changes in the data set if the data are collected through time
 - ▶ **Generating new data**: closely related to the previous point.



1. We have seen discriminative models
 - ▶ Given an image x , predict label y
 - ▶ Estimates $P(y|x)$
2. Discriminative models have several key limitations
 - ▶ Can't model $P(x)$, i.e. the probability of seeing a certain image
 - ▶ Thus, can't sample from $P(x)$, i.e. can't generate new images
3. Generative models (in general) cope with all of the above problems
 - ▶ Can model $P(x)$
 - ▶ Can generate x such as new images
4. Generate new data by sampling from the learned distribution.
5. Evaluate the likelihood of data observed at test time.
6. Find the conditional relationship between variables, eg learning the distribution $p(x_2|x_1)$ allows us to build discriminative classification or regression models.
7. Score algorithms by using complexity measures like entropy, mutual information, and moments of the distribution.

1. Given training data, generate new samples from same distribution¹,



Train from $x \sim p_{data}(x)$

Generate from $x \sim p_{model}(x)$

Want to learn $p_{model}(x)$ similar to $p_{data}(x)$

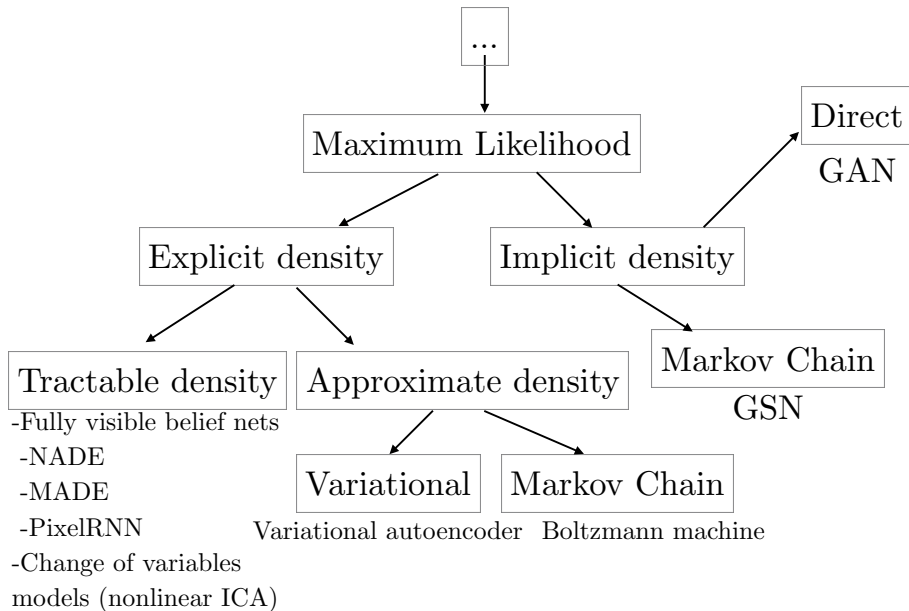
2. Several flavors

- ▶ **Explicit density estimation:** explicitly define and solve for $p_{model}(x)$
- ▶ **Implicit density estimation:** learn model that can sample from $p_{model}(x)$ w/o explicitly defining it

¹Taken from Fei-Fei Li et al. slides and Tutorial on Generative Adversarial Networks, 2017.

1. The following images were generated from a generative model (Karras et al. 2018).







1. Without using latent variables
 - ▶ Parametric density estimation
 - ▶ Non parametric density estimation
2. With using latent variables
 - ▶ Mixture models
 - ▶ Deep generative models

Introduction

Generative models without using latent variables



1. We assume x_1, x_2, \dots, x_m are IID random variables distributed as $p(x; \theta)$, hence we have

$$p(x; \theta) = p(x_1, x_2, \dots, x_m; \theta) = \prod_{k=1}^m p(x_k; \theta)$$

2. $p(x; \theta)$ is a function of θ and is known as **likelihood function**.
3. The **maximum likelihood (ML)** method estimates θ so that the likelihood function takes its maximum value, that is,

$$\hat{\theta}_{ML} = \underset{\theta}{\operatorname{argmax}} \prod_{k=1}^m p(x_k; \theta)$$

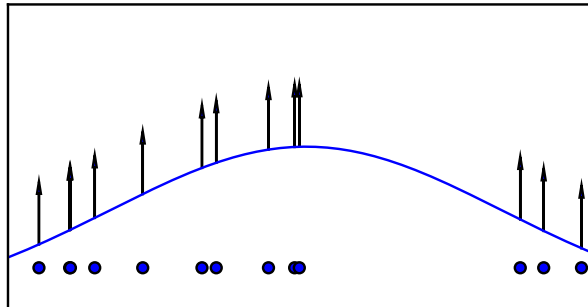
4. To obtain $\hat{\theta}_{ML}$ that maximizing the likelihood function, we must have

$$\frac{\partial \prod_{k=1}^m p(x_k; \theta)}{\partial \theta} = 0$$



1. It is more convenient to work with the logarithm of the likelihood function than with the likelihood function itself. Hence,

$$\mathcal{L}(\theta) = \ln \prod_{k=1}^m p(x_k; \theta) = \sum_{k=1}^m \ln p(x_k; \theta)$$



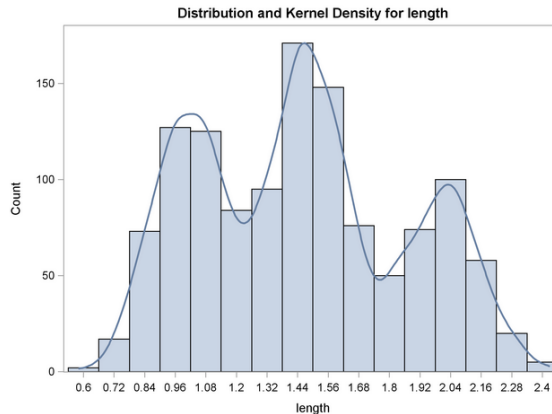
$$\hat{\theta}_{ML} = \underset{\theta}{\operatorname{argmax}} \mathcal{L}(\theta)$$



1. Parametric forms do not always fit the densities encountered in practice.
2. Most of parametric densities are unimodal, whereas many practical problems have multi-modal densities.
3. Non-parametric methods can be used with arbitrary distributions without assumption of knowing the forms of the underlying densities.
4. In nonparametric estimation, we assume that **similar inputs have similar outputs**.
5. This is a reasonable assumption because **the world is smooth** and functions, whether they are densities, discriminants, or regression functions, change slowly.
6. Some approaches for nonparametric density estimation
 - ▶ Histogram
 - ▶ Parzen window
 - ▶ Kernel density estimator
 - ▶ Nearest neighbors

1. Divide the space into a set of regular intervals of the form

$$I_j = (x_0 + jh, x_0 + (j + 1)h], \quad \text{for } j \in \{\dots, -1, 0, 1, \dots\}.$$



2. In each interval, the density is constant and is proportional to the number of observations falling into this interval.



1. **Naive estimator**, addresses the **choice of bin locations**, thus the origin is eliminated.
2. For bin width h , bin denoted by $\mathcal{R}(x)$ is interval $[x - \frac{h}{2}, x + \frac{h}{2})$ and the estimate is

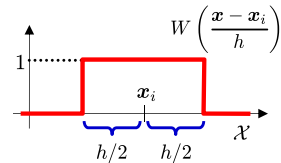
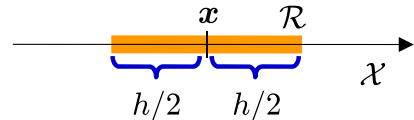
$$\hat{p}(x) = \frac{|\mathcal{R}(x)|}{mh}$$

3. The estimator can also be written as

$$\hat{p}(x) = \frac{1}{mh} \sum_{k=1}^m w\left(\frac{x - x_k}{h}\right)$$

w is **weight function** and defined as

$$w(u) = \begin{cases} 1 & \text{if } |u| \leq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases}$$





1. To get a smooth estimate, a smooth weight function (**kernel function**) is used.

$$\hat{p}(x) = \frac{1}{mh} \sum_{i=1}^m w\left(\frac{x - x_i}{h}\right)$$

$w(\cdot)$ is some **kernel function** and h is the **smoothing parameter**.

2. **Gaussian kernel function** with mean 0 and variance 1 is usually used.

$$w(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u^2}{2}\right)$$

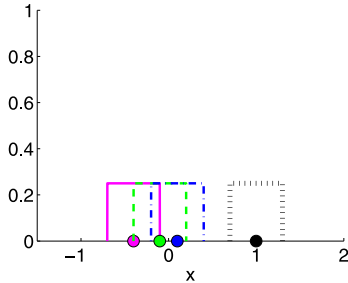
3. Function $w(\cdot)$ determines shape of influences and h determines window width.
4. The kernel estimator can be generalized to D -dimensional data.

$$\hat{p}(x) = \frac{1}{mh^D} \sum_{k=1}^m w\left(\frac{x - x_k}{h}\right)$$

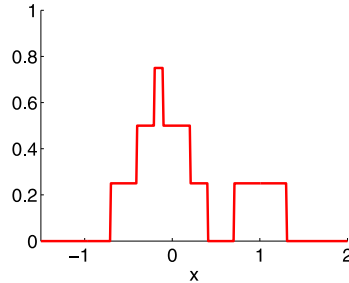
$$w(u) = \left(\frac{1}{\sqrt{2\pi}}\right)^D \exp\left(-\frac{\|u\|^2}{2}\right)$$

5. The total number of data points lying in this window (cube) equals to (**drive it.**)

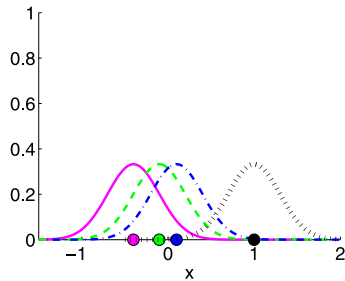
$$k = \sum_{i=1}^m w\left(\frac{x - x_i}{h}\right)$$



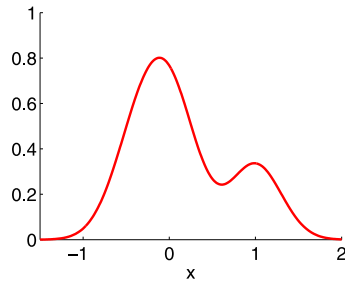
(a) Each Parzen window function



(b) Parzen window estimator



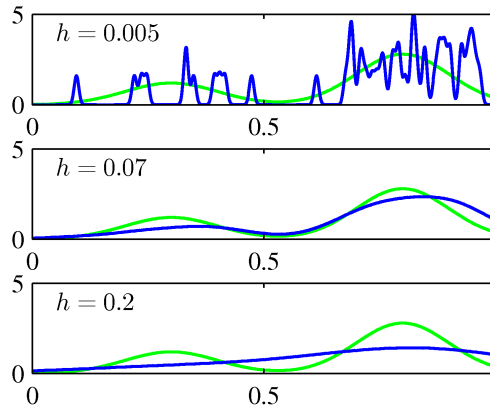
(a) Each Gaussian kernel function



(b) Kernel density estimator



- ▶ A difficulty with KDE is that the parameter h is fixed for all kernels.
- ▶ Large value of h may lead to **over-smoothing**.
- ▶ Reducing value of h may lead to **noisy estimates**.
- ▶ The **optimal choice of h** may be dependent on **location within the data space**.





- ▶ Instead of fixing h and determining the value of k from the data, we fix the value of k and use the data to find an appropriate value of h .
- ▶ To do this, we consider a small sphere centered on the point x at which we wish to estimate the density $p(x)$ and allow the radius of the sphere to grow until it contains precisely k data points (Why?).

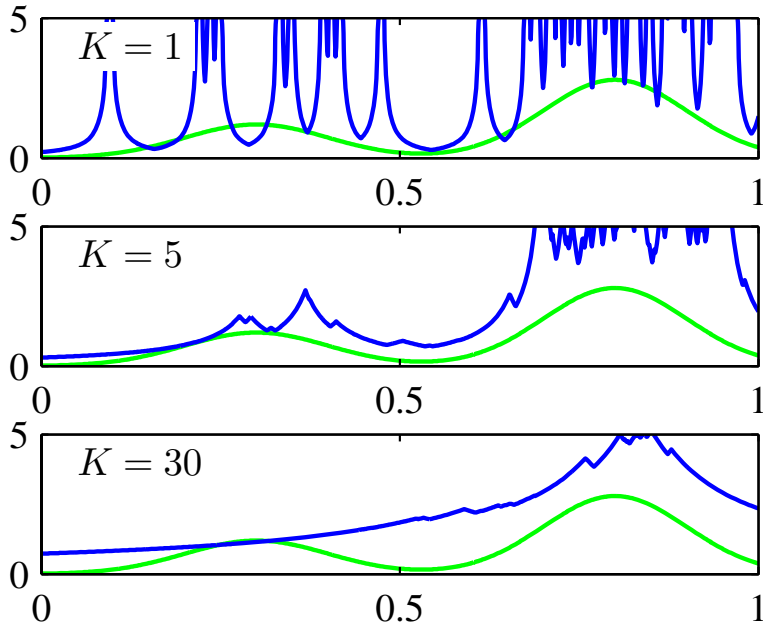
$$\hat{p}(x) = \frac{k}{mV}$$

V is the volume of the resulting sphere.

- ▶ Value of k determines the degree of smoothing and there is an optimum choice for k that is neither too large nor too small.
- ▶ **Note that:** The model produced by k nearest neighborhood **is not a true density model** because the **integral over all space diverges**.

Theorem

It can be shown that both the K -NN and the kernel density estimators converge to the true probability density in the limit $N \rightarrow \infty$ provided V shrinks suitably with N , and K grows with N (Duda, Hart, and Stork 2001).



Introduction

Generative models using latent variables



1. An alternative way to model an unknown density function $p(x)$ is via linear combination of M density functions in the form of

$$p(x) = \sum_{k=1}^M \pi_k p(x|k)$$

where

$$\sum_{k=1}^M \pi_k = 1$$
$$\int_x p(x|k) dx = 1$$

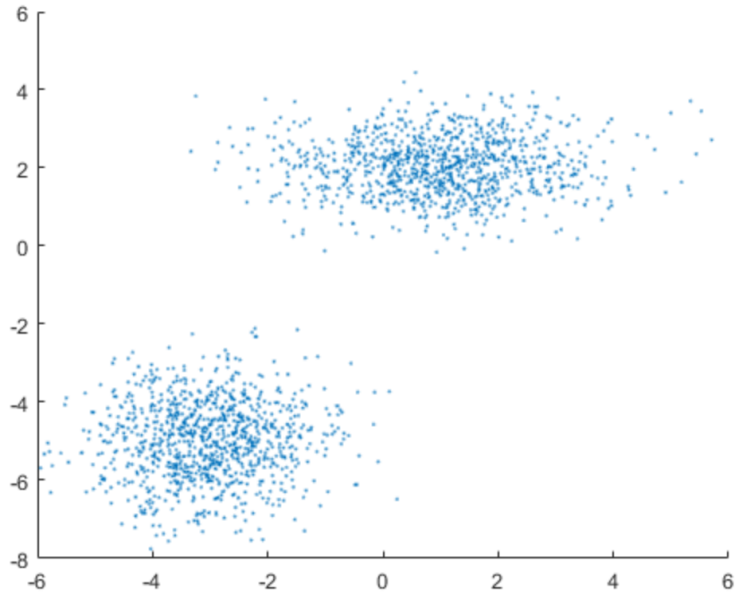
2. This modeling implicitly assumes that each point x may be drawn from any M model distributions with probability π_k (for $k = 1, 2, \dots, M$).



1. It can be shown that this modeling can approximate closely any continuous density function for a sufficient number of mixtures M and appropriate model parameters.
2. First, we select a set of density components $p(x|k)$ in the parametric form $p(x|k, \theta)$.

$$p(x; \theta) = \sum_{k=1}^M \pi_k p(x|\theta_k)$$

3. Then, we compute parameters $\theta_1, \theta_2, \dots, \theta_M$ and $\pi_1, \pi_2, \dots, \pi_M$ based on training data.
4. The parameter set is defined as $\theta = \{\pi_1, \pi_2, \dots, \pi_M, \theta_1, \theta_2, \dots, \theta_M\}$ and $\sum_i \pi_i = 1$.
5. In order to find parameters, we use **EM** algorithm.



Deep generative models



1. We assume that dataset $S = \{x_1, x_2, \dots, x_m\}$ are samples from distribution $p(x)$.
2. Goal of any generative model is to approximate $p(x)$ given access to the dataset S .
3. If we can learn a good generative model, we can use it for inference.
4. We usually have three fundamental inference queries for evaluating a generative model.
 - ▶ **Density estimation:** Given a point x , what is the probability assigned by the model, i.e., $p(x; \theta)$?
 - ▶ **Sampling:** How can we generate new data from the model distribution, i.e., $x_{new} \sim p(x; \theta)$?
 - ▶ **Unsupervised representation learning:** How can we learn meaningful feature representations for a point x ?

Deep generative models

Latent variable models

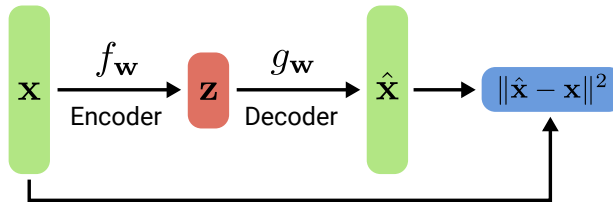


1. These models map between **observation space** $\mathbf{x} \in \mathbb{R}^D$ and **latent space** $\mathbf{z} \in \mathbb{R}^Q$:

Encoder $f_{\mathbf{w}} = \mathbf{x} \mapsto \mathbf{z}$

Decoder $g_{\mathbf{w}} = \mathbf{z} \mapsto \hat{\mathbf{x}}$

2. Each latent variable gets associated with each data point in the training set.
3. The latent vectors are smaller than the observations ($Q < D$) \Rightarrow compression.
4. Models are linear or non-linear, deterministic or stochastic, with/without encoder.



²This slide taken from slides of Prof. Geiger



1. These models often consider a simple Bayesian model

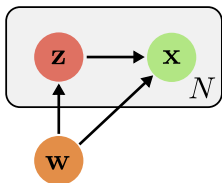
$$p(\mathbf{x}) = \int_{\mathbf{z}} p(\mathbf{z})p(\mathbf{x}|\mathbf{z})d\mathbf{z} = \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} [p(\mathbf{x}|\mathbf{z})]$$

- ▶ $p(\mathbf{z})$ is the prior over the latent variable $\mathbf{z} \in \mathbb{R}^Q$.
 - ▶ $p(\mathbf{x}|\mathbf{z})$ is the likelihood (= decoder that transforms \mathbf{z} into a distribution over \mathbf{x}).
 - ▶ $p(\mathbf{x})$ is the marginal over the joint distribution $p(\mathbf{x}, \mathbf{z})$.
2. The goal is to maximize $p(\mathbf{x})$ for dataset S by learning the two models $p(\mathbf{z})$ and $p(\mathbf{x}|\mathbf{z})$ such that latent variables \mathbf{z} best capture the latent structure of data.

³This slide taken from slides of Prof. Geiger



1. These models are represented using **graphical model in plate notation**.

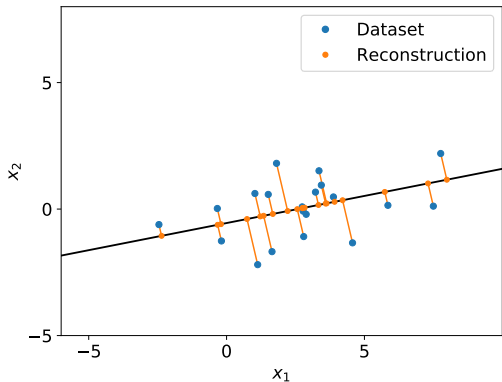


- ▶ Variables inside plates are replicated (we have N data points to explain).
- ▶ Each data point x is associated with a latent variable z .
- ▶ We use a single w to refer to all model parameters and parameters are global (exist only once).

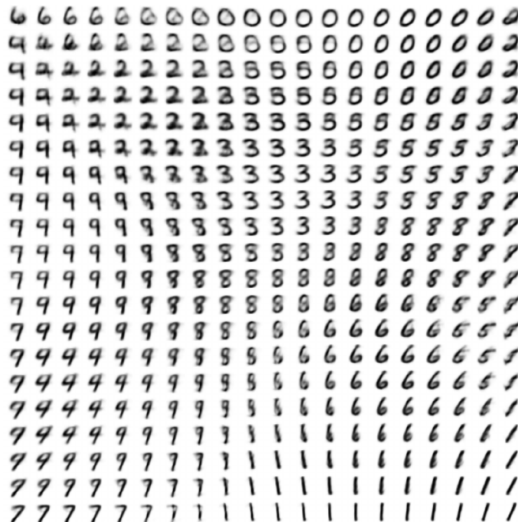
⁴This slide taken from slides of Prof. Geiger



1D Manifold in 2D Space



Learned MNIST manifold



Deep generative models

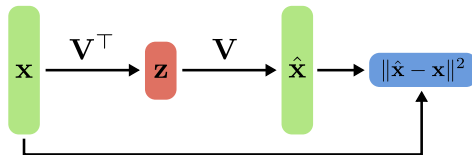
Principal component analysis



- Let
 - ▶ $\mathbf{X} = (\mathbf{x}_1, \dots, \mathbf{x}_N)^T \in \mathbb{R}^{N \times D}$ be a dataset of samples $\mathbf{x}_i \in \mathbb{R}^D$, and
 - ▶ $\mathbf{Z} = (\mathbf{z}_1, \dots, \mathbf{z}_N)^T \in \mathbb{R}^{N \times Q}$ be the corresponding latent variables $\mathbf{z}_i \in \mathbb{R}^Q$.
- The goal of PCA is to learn a **linear bidirectional mapping** $\mathcal{X} \longleftrightarrow \mathcal{Z}$ such that as much information of \mathcal{X} as possible is retained in \mathcal{Z} .
- Let the following **linear mapping** maps data from **latent** to **observation** space.

$$\hat{\mathbf{x}}_i = \bar{\mathbf{x}} + \sum_{j=1}^Q z_{ij} \mathbf{v}_j$$

where $\bar{\mathbf{x}}$ is **data mean** and $\mathbf{V} = (\mathbf{v}_1, \dots, \mathbf{v}_Q)$ is an **orthonormal basis**.



- The goal is to minimize the **L_2 reconstruction loss** wrt. \mathbf{Z} and \mathbf{V} .

$$\mathcal{L}(\mathbf{Z}, \mathbf{V}) = \sum_{i=1}^N \|\hat{\mathbf{x}}_i - \mathbf{x}_i\|^2$$

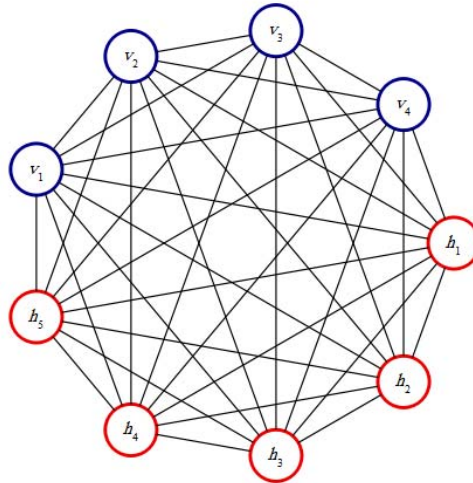
⁵This slide taken from slides of Prof. Geiger

Deep generative models

Boltzmann Machine



1. BMs are fully connected networks of binary units.
2. BM is an undirected symmetric network of binary units that are divided into **visible** and **hidden** units.





1. BMs are theoretically capable of **learning any given distribution**.
2. The network **sets the strengths of the connections between the units** to capture the **correlations** between them to build a generative network **capable of producing new examples of the same distribution**.
3. Since all variables in a BM are not directly observed, it gives us a handle to control the sampling of new examples.
4. The model can take in an incomplete example and use it to output the complete version.



1. BM is a network with an **energy** defined for the overall network.
2. For a BM with only observed units, the energy is defined as

$$\begin{aligned} E(\mathbf{x}) &= - \sum_{ij} w_{ij} x_i x_j - \sum_{i=1} b_i x_i \\ &= -\mathbf{x}^\top \mathbf{W} \mathbf{x} - \mathbf{b}^\top \mathbf{x} \end{aligned}$$

$H(\mathbf{x}) = -E(\mathbf{x})$ **Alternatively, happiness is used to avoid multiple minus signs.**

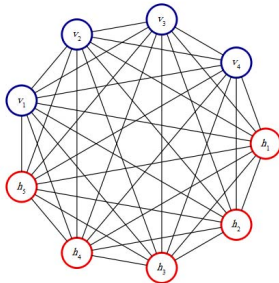
- ▶ $\mathbf{x} = (x_1, x_2, \dots, x_d) \in \{0, 1\}^d$ is the input vector.
 - ▶ $\mathbf{W} = (w_{ij})$ is the weight matrix
 - ▶ $\mathbf{b} = (b_1, b_2, \dots, b_d) \in \{0, 1\}^d$ is the bias vector.
3. The joint probability distribution defined as

$$p_{\text{model}}(\mathbf{x}) = \frac{\exp(-E(\mathbf{x}))}{Z}$$

Z is **Partition function** that ensures $\sum_{\mathbf{x}} p_{\text{model}}(\mathbf{x}) = 1$.



1. BM becomes more powerful when not all the variables are observed.
2. The **latent variables** can act similarly to **hidden units** in a MLP.



3. By decomposing units into two subsets: **visible \mathbf{v}** and **hidden units \mathbf{h}** , we obtain.

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{v}^\top \mathbf{R} \mathbf{v} - \mathbf{v}^\top \mathbf{W} \mathbf{h} - \mathbf{h}^\top \mathbf{S} \mathbf{h} - \mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h}$$

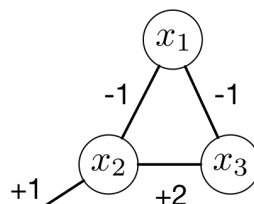
4. The joint probability distribution defined as

$$p_{\text{model}}(\mathbf{v}, \mathbf{h}) = \frac{\exp(-E(\mathbf{v}, \mathbf{h}))}{Z}$$

Z is **Partition function** that ensures $\sum_{\mathbf{x}} p_{\text{model}}(\mathbf{x}) = 1$.



Example:



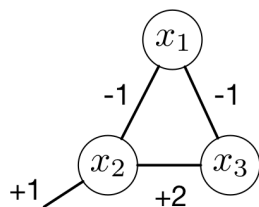
x_1	x_2	x_3	$w_{12}x_1x_2$	$w_{13}x_1x_3$	$w_{23}x_2x_3$	b_2x_2	$H(\mathbf{x})$	$\exp(H(\mathbf{x}))$	$p(\mathbf{x})$
-1	-1	-1	-1	-1	2	-1	-1	0.368	0.0021
-1	-1	1	-1	1	-2	-1	-3	0.050	0.0003
-1	1	-1	1	-1	-2	1	-3	0.368	0.0021
-1	1	1	1	1	2	1	5	148.413	0.8608
1	-1	-1	1	1	2	-1	3	20.086	0.1165
1	-1	1	1	-1	-2	-1	-3	0.050	0.0003
1	1	-1	-1	1	-2	1	-1	0.368	0.0021
1	1	1	-1	-1	2	1	1	2.718	0.0158

$$\mathcal{Z} = 172.420$$



Marginal probabilities:

$$\begin{aligned}
 p(x_1 = 1) &= \frac{1}{Z} \sum_{\mathbf{x}: x_1=1} \exp(H(\mathbf{x})) \\
 &= \frac{20.086 + 0.050 + 0.368 + 2.718}{172.420} \\
 &= 0.135
 \end{aligned}$$



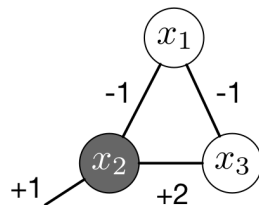
x_1	x_2	x_3	$w_{12}x_1x_2$	$w_{13}x_1x_3$	$w_{23}x_2x_3$	b_2x_2	$H(\mathbf{x})$	$\exp(H(\mathbf{x}))$	$p(\mathbf{x})$
-1	-1	-1	-1	-1	2	-1	-1	0.368	0.0021
-1	-1	1	-1	1	-2	-1	-3	0.050	0.0003
-1	1	-1	1	-1	-2	1	-3	0.368	0.0021
-1	1	1	1	1	2	1	5	148.413	0.8608
1	-1	-1	1	1	2	-1	3	20.086	0.1165
1	-1	1	1	-1	-2	-1	-3	0.050	0.0003
1	1	-1	-1	1	-2	1	-1	0.368	0.0021
1	1	1	-1	-1	2	1	1	2.718	0.0158

$$Z = 172.420$$



Conditional probabilities:

$$\begin{aligned}
 p(x_1 = 1 | x_2 = -1) &= \frac{\sum_{\mathbf{x}: x_1=1, x_2=-1} \exp(H(\mathbf{x}))}{\sum_{\mathbf{x}: x_2=-1} \exp(H(\mathbf{x}))} \\
 &= \frac{20.086 + 0.050}{0.368 + 0.050 + 20.086 + 0.050} \\
 &= 0.980
 \end{aligned}$$



x_1	x_2	x_3	$w_{12}x_1x_2$	$w_{13}x_1x_3$	$w_{23}x_2x_3$	b_2x_2	$H(\mathbf{x})$	$\exp(H(\mathbf{x}))$	$p(\mathbf{x})$
-1	-1	-1	-1	-1	2	-1	-1	0.368	0.0021
-1	-1	1	-1	1	-2	-1	-3	0.050	0.0003
-1	1	-1	1	-1	-2	1	-3	0.368	0.0021
-1	1	1	1	1	2	1	5	148.413	0.8608
1	-1	-1	1	1	2	-1	3	20.086	0.1165
1	-1	1	1	-1	-2	-1	-3	0.050	0.0003
1	1	-1	-1	1	-2	1	-1	0.368	0.0021
1	1	1	-1	-1	2	1	1	2.718	0.0158



1. Learning algorithms for BMs are usually based on **maximum likelihood**.
2. All BMs have an **intractable partition function**, so the **maximum likelihood gradient must be approximated**.
3. An interesting property of BMs is that the update for a particular w_{ij} depends only on the statistics of x_i and x_j .



1. A BM admits the following likelihood for points $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$.

$$\mathcal{L}(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}) = \prod_{i=1}^n p(\mathbf{x}^{(i)})$$

2. We will work with the log-likelihood instead of the true likelihood.

$$\begin{aligned} \log \mathcal{L}(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}) &= \sum_{k=1}^n \log \frac{\exp(H(\mathbf{x}^{(k)}))}{Z} \\ &= \sum_{k=1}^n \log(\exp(H(\mathbf{x}^{(k)}))) - \log Z \\ &= \sum_{k=1}^n H(\mathbf{x}^{(k)}) - \log Z \end{aligned}$$

3. The aim is to maximize $\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\mathcal{L}(\mathbf{x})]$

$$\mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [\mathcal{L}(\mathbf{x})] = \sum_{k=1}^n p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) \mathcal{L}(\mathbf{x}^{(k)})$$



1. Now, deriving the gradient with respect to the weights ($\nabla_{w_{i,j}} \log \mathcal{L}$)

$$\begin{aligned} \nabla_{w_{i,j}} \left[\sum_{k=1}^n p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) \left(H(\mathbf{x}^{(k)}) - \log Z \right) \right] &= \sum_{k=1}^n p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) \nabla_{w_{i,j}} H(\mathbf{x}^{(k)}) \\ &\quad - \sum_{k=1}^n p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) \nabla_{w_{i,j}} \log Z \end{aligned}$$

2. The first term equals to

$$\begin{aligned} \sum_{k=1}^n p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) \nabla_{w_{i,j}} H(\mathbf{x}^{(k)}) &= \sum_{k=1}^n p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) \nabla_{w_{i,j}} \left[\sum_{i \neq j} w_{i,j} x_i^{(k)} x_j^{(k)} \right. \\ &\quad \left. + \sum_i p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) b_i x_i^{(k)} \right] \\ &= \sum_{k=1}^n p_{\text{data}}(\mathbf{x} = \mathbf{x}^{(k)}) x_i^{(k)} x_j^{(k)} \\ &= \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [x_i x_j] \end{aligned}$$



1. The second term equals to

$$\begin{aligned}\nabla_{w_{i,j}} \log Z &= \nabla_{w_{i,j}} \log \sum_{\mathbf{x}} \exp(H(\mathbf{x})) \\ &= \frac{1}{\sum_{\mathbf{x}} \exp(H(\mathbf{x}))} \nabla_{w_{i,j}} \sum_{\mathbf{x}} \exp(H(\mathbf{x})) = \frac{1}{Z} \nabla_{w_{i,j}} \sum_{\mathbf{x}} \exp(H(\mathbf{x})) \\ &= \frac{1}{Z} \sum_{\mathbf{x}} \exp(H(\mathbf{x})) \nabla_{w_{i,j}} H(\mathbf{x}) = \sum_{\mathbf{x}} \frac{\exp(H(\mathbf{x}))}{Z} \nabla_{w_{i,j}} H(\mathbf{x}) \\ &= \sum_{\mathbf{x}} p_{\text{model}}(\mathbf{x}) \nabla_{w_{i,j}} H(\mathbf{x}) \\ &= \sum_{\mathbf{x}} p_{\text{model}}(\mathbf{x}) [x_i x_j] \\ &= \mathbb{E}_{\mathbf{x} \sim p_{\text{model}}} [x_i x_j]\end{aligned}$$



1. By combining the above equations, the gradient w.r.t weights becomes

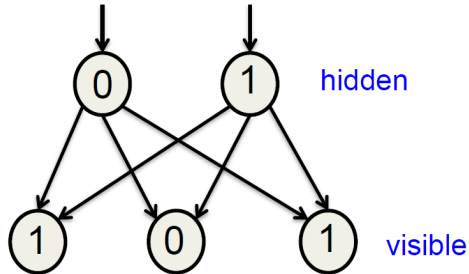
$$\nabla_{w_{i,j}} \log \mathcal{L} = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [x_i x_j] - \mathbb{E}_{\mathbf{x} \sim p_{\text{model}}} [x_i x_j]$$

2. By combining the above equations, the gradient w.r.t biases becomes

$$\nabla_{b_i} \log \mathcal{L} = \mathbb{E}_{\mathbf{x} \sim p_{\text{data}}} [x_i] - \mathbb{E}_{\mathbf{x} \sim p_{\text{model}}} [x_i]$$



1. In BM, we generate in two steps:
 - ▶ Pick the hidden states from $p(\mathbf{h})$.
 - ▶ Pick the visible states from $p(\mathbf{v}|\mathbf{h})$.



2. The probability of generating a visible vector, \mathbf{v} , is computed by summing over all possible hidden states.

$$p(\mathbf{v}) = \sum_{\mathbf{h}} p(\mathbf{h})p(\mathbf{v}|\mathbf{h})$$



1. Given an ordered set of variable, x_1, \dots, x_d , and a starting configuration $x^0 = (x_1^0, \dots, x_d^0)$,

Gibbs sampling uses the following procedure

- ▶ Repeat until convergence for $t = 1, 2, \dots$,
 - ▶ Set $\mathbf{x} \leftarrow \mathbf{x}^{t-1}$.
 - ▶ For each variable x_i in the order we fixed:
 - 1) Sample $x'_i \sim p(x_i \mid \mathbf{x}_{-i})$.
 - 2) Update $\mathbf{x} \leftarrow (x_1, \dots, x'_i, \dots, x_d)$.
 - ▶ Set $\mathbf{x}^t \leftarrow \mathbf{x}$.

We use \mathbf{x}_{-i} to denote all variables in \mathbf{x} except x_i .

2. It is often very easy to performing each sampling step, since we only need to condition x_i on other variables.
3. Note that when we update x_i , we immediately use its new value for sampling other variables x_j .



1. We derive $p(x_i | \mathbf{x}_{-i})$ using probability of axioms and discarding bias terms

$$\begin{aligned} p(x_i = 1 | \mathbf{x}_{-i}) &= \frac{p(x_i = 1, \mathbf{x}_{-i})}{p(x_i = 1, \mathbf{x}_{-i}) + p(x_i = 0, \mathbf{x}_{-i})} \\ &= \frac{\exp \left[\sum_{j \neq i} w_{ij} x_j \right]}{1 + \exp \left[\sum_{j \neq i} w_{ij} x_j \right]} \\ &= \frac{1}{1 + \exp \left[- \sum_{j \neq i} w_{ij} x_j \right]} \\ &= \sigma \left(\sum_{j \neq i} w_{i,j} x_j \right) \end{aligned}$$



1. Let $d = 3$, we need to define

$$x'_0 \sim p(x_0 | x_1, x_2)$$

$$x'_1 \sim p(x_1 | x'_0, x_2)$$

$$x'_2 \sim p(x_2 | x'_0, x'_1)$$

2. Each dimension is binary, the above 3 models must necessarily return the probability of observing **a 1**.
3. Note that when we update x_i , we immediately use its new value for sampling other variables x_j .



1. We derive $p(x_0|x_1, x_2)$ using probability of axioms

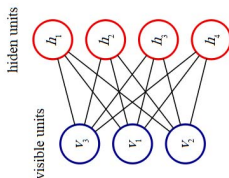
$$\begin{aligned}
 p(x_0 = 1|x_1, x_2) &= \frac{p(x_0 = 1, x_1, x_2)}{p(x_1, x_2)} = \frac{p(x_0 = 1, x_1, x_2)}{\sum_{x_0 \in \{0,1\}} p(x_0, x_1, x_2)} \\
 &= \frac{p(x_0 = 1, x_1, x_2)}{p(x_0 = 0, x_1, x_2) + p(x_0 = 1, x_1, x_2)} \\
 &= \frac{1}{1 + \frac{p(x_0=0, x_1, x_2)}{p(x_0=1, x_1, x_2)}} = \frac{1}{1 + \frac{\exp(H(x_0=0, x_1, x_2))}{\exp(H(x_0=1, x_1, x_2))}} \\
 &= \frac{1}{1 + \exp(H(x_0 = 0, x_1, x_2) - H(x_0 = 1, x_1, x_2))} \\
 &= \frac{1}{1 + \exp(\sum_{i \neq j} w_{ij} x_i x_j + \sum_i b_i x_i - (\sum_{i \neq j} w_{ij} x_i x_j + \sum_i b_i x_i))} \\
 &= \frac{1}{1 + \exp(-\sum_{j \neq i=0} w_{ij} x_j - b_i)} \\
 &= \sigma\left(\sum_{j \neq i=0} w_{i,j} x_j + b_i\right)
 \end{aligned}$$

Deep generative models

Restricted Boltzmann Machine



1. The **tractability** of the joint distribution is one of the **biggest drawbacks of BMs**.
2. RBMs are a special type of BMs with two layers: **One visible** and **one hidden** layer.



3. The connections in an RBM are **undirected** and the graph is a **bipartite graph**.
4. By the Markov property, $p(\mathbf{h}|\mathbf{v})$ and $p(\mathbf{v}|\mathbf{h})$ both factorize (**Show later**).

$$p(\mathbf{h}|\mathbf{v}) = \prod_i p(h_i|\mathbf{v})$$

$$p(\mathbf{v}|\mathbf{h}) = \prod_j p(v_j|\mathbf{h})$$

5. There is no need for **variational Bayes** and **Gibbs sampling** can be implemented efficiently by alternating between hidden and visible levels, known as **block Gibbs sampling**.
6. The marginal distributions $p(\mathbf{v})$ and $p(\mathbf{h})$ do not factorize (**Show it**).



1. This bipartite architecture allows us to have more control over the joint distribution.
2. RBMs are a powerful replacement for fully connected BMs when building a deep architecture because of the independence of units within the same layer, which allows for more freedom and flexibility.
3. The **latent variables** can act similarly to **hidden units** in a MLP.
4. RBMs can be trained using the techniques of **maximum likelihood**.
5. Sampling from an RBM can be done using **Gibbs sampling** method or any other **Markov Chain Monte Carlo (MCMC)** method.



1. Hidden units are conditionally **independent** given the **visible units** and **vice versa**.

$$p(v_i = 1 | \mathbf{h}) = \sigma \left(\sum_j w_{ij} h_j + b_i \right)$$

$$p(h_j = 1 | \mathbf{v}) = \sigma \left(\sum_i w_{ij} v_i + c_j \right)$$

2. Given **visible \mathbf{v}** , we can sample each **h** independently.
3. Given **hidden \mathbf{h}** , we can sample each **v_j** independently.



1. The energy of the joint state $\{\mathbf{v}, \mathbf{h}\}$ is defined as follows:

$$E(\mathbf{v}, \mathbf{h}; \theta) = -\mathbf{v}^\top \mathbf{W} \mathbf{h} - \mathbf{b}^\top \mathbf{v} - \mathbf{a}^\top \mathbf{h}$$

where $\theta = \{\mathbf{W}, \mathbf{b}, \mathbf{a}\}$ are the model parameters. W_{ij} represents the symmetric interaction term between visible variable i and hidden variable j , and b_i and a_j are bias terms.

2. The joint distribution equals to

$$p(\mathbf{v}, \mathbf{h}; \theta) = \frac{1}{Z(\theta)} \exp(-E(\mathbf{v}, \mathbf{h}; \theta))$$
$$Z(\theta) = \sum_{\mathbf{v}} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}; \theta))$$



1. The model assigns the following probability to a visible vector \mathbf{v}

$$p(\mathbf{v}; \theta) = \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}; \theta)$$

2. The hidden variables can be explicitly marginalized out

$$\begin{aligned} p(\mathbf{v}; \theta) &= \frac{1}{Z(\theta)} \sum_{\mathbf{h}} \exp(-E(\mathbf{v}, \mathbf{h}; \theta)) \\ &= \frac{1}{Z(\theta)} \sum_{\mathbf{h}} \exp\left(\mathbf{v}^\top \mathbf{W} \mathbf{h} + \mathbf{b}^\top \mathbf{v} + \mathbf{a}^\top \mathbf{h}\right) \\ &= \frac{1}{Z(\theta)} \exp(\mathbf{b}^\top \mathbf{v}) \prod_{j=1}^F \sum_{h_j \in \{0,1\}} \exp\left(a_j h_j + \sum_i W_{ij} v_i h_j\right) \\ &= \frac{1}{Z(\theta)} \exp(\mathbf{b}^\top \mathbf{v}) \prod_{j=1}^F \left(1 + \exp\left(a_j + \sum_i W_{ij} v_i\right)\right) \end{aligned}$$



1. Bipartite graph structure of RBM has the following property.
2. Conditionals $p(\mathbf{h}|\mathbf{v})$ and $p(\mathbf{v}|\mathbf{h})$ are factorized and easy computed.

$$\begin{aligned}
 p(\mathbf{h}|\mathbf{v}) &= \frac{p(\mathbf{h}, \mathbf{v})}{p(\mathbf{v})} = \frac{1}{p(\mathbf{v})} \frac{1}{Z} \exp\left(\mathbf{b}^\top \mathbf{v} + \mathbf{c}^\top \mathbf{h} + \mathbf{v}^\top \mathbf{W} \mathbf{h}\right) \\
 &= \frac{1}{Z'} \exp\left(\mathbf{c}^\top \mathbf{h} + \mathbf{v}^\top \mathbf{W} \mathbf{h}\right) \\
 &= \frac{1}{Z'} \exp\left(\sum_j c_j h_j + \sum_j \mathbf{v}^\top \mathbf{W}_{:j} h_j\right) \\
 &= \frac{1}{Z'} \prod_j \exp\left(c_j h_j + \mathbf{v}^\top \mathbf{W}_{:j} h_j\right)
 \end{aligned}$$

3. Normalizing the distributions over individual binary h

$$\begin{aligned}
 p(h_j = 1|\mathbf{v}) &= \frac{\tilde{p}(h_j = 1|\mathbf{v})}{\tilde{p}(h_j = 0|\mathbf{v}) + \tilde{p}(h_j = 1|\mathbf{v})} \\
 &= \frac{\exp(c_j + \mathbf{v}^\top \mathbf{W}_{:j})}{\exp(0) + \exp(c_j + \mathbf{v}^\top \mathbf{W}_{:j})} = \sigma\left(c_j + \mathbf{v}^\top \mathbf{W}_{:j}\right)
 \end{aligned}$$

4. Similarly

$$p(v_i = 1|\mathbf{h}) = \sigma(c_i + \mathbf{W}_i \cdot \mathbf{h})$$



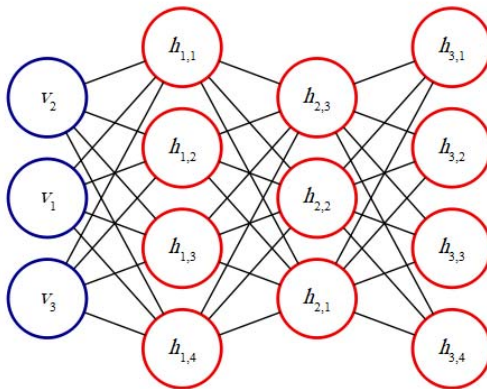
1. **Step 1:** Take input vector to the visible node
2. **Step 2:** Update the weights of all hidden nodes in parallel given the current states of the units in the other layer.
3. **Step 3:** Reconstruct the input vector with the same weights used for hidden nodes. Even though we use the same weights, the reconstructed input will be different as multiple hidden nodes contribute the reconstructed input.
4. **Step 4:** Compare the input to the reconstructed input based on KL divergence.
5. **Step 5:** Reconstruct the input vector again and keep repeating for all the input data and for multiple epochs. This is repeated until the system is in equilibrium distribution.

Deep generative models

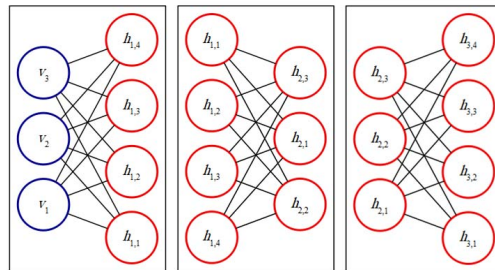
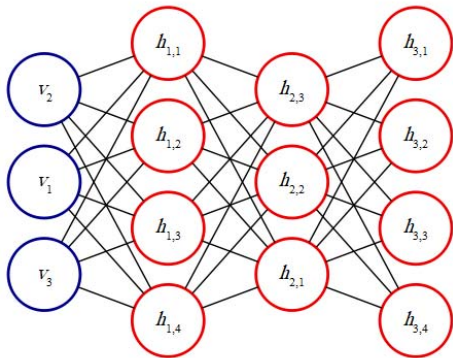
Deep Boltzmann Machine



1. DBM is an undirected deep network of **several hidden layers** (Salakhutdinov and Larochelle 2010).
2. Every **unit is connected** to every unit from the **adjacent layers**.
3. There are **no connections** between **units of the same layer**.



1. DBMs can also be viewed as a group of RBMs stacked together.

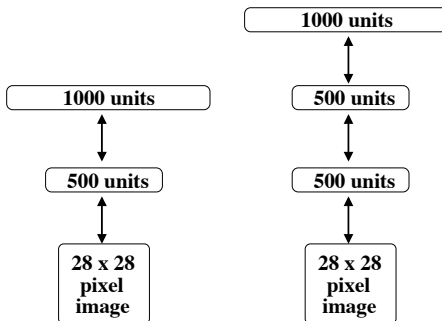


2. Training of DBMs is often done in two stages:

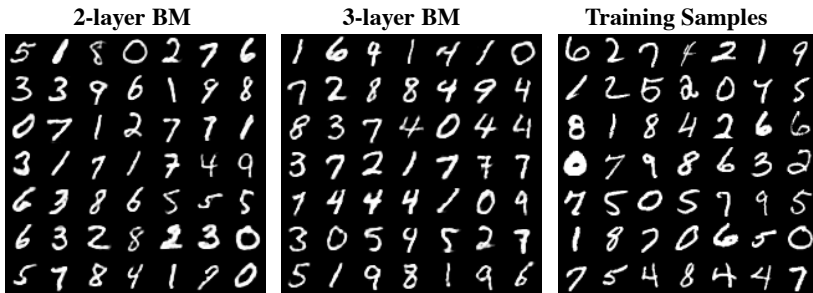
- ▶ A **pre-training stage** where every RBM is trained independently.
- ▶ a **fine tuning stage** where the network is trained at once using backpropagation.



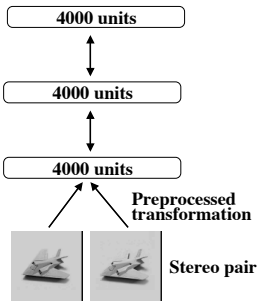
1. Considering two architectures for MNIST dataset.



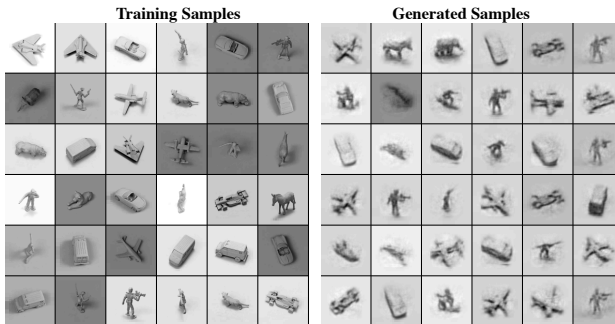
2. The results using Gibbs sampling.



1. Considering the following architecture for NORB dataset.



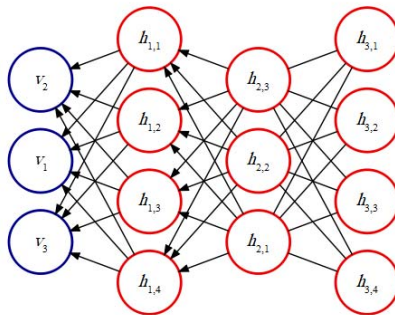
2. The results using Gibbs sampling.



Deep generative models

Deep Belief Networks

1. DBN is a hybrid PGM involving both directed and undirected connections.
2. Deep belief networks consisting of many hidden layers.
3. Connections between top two layers are undirected
4. Connections between all other layers is directed, pointing towards data.



$$p(\mathbf{v}, \mathbf{h}^{(1)}, \mathbf{h}^{(2)}, \dots, \mathbf{h}^{(k)}) = p(\mathbf{v}|\mathbf{h}^{(1)})p(\mathbf{h}^{(1)}|\mathbf{h}^{(2)}) \dots p(\mathbf{h}^{(k-2)}|\mathbf{h}^{(k-1)})p(\mathbf{h}^{(k-1)}, \mathbf{h}^{(k)})$$

5. $p(\mathbf{h}^{(k-1)}, \mathbf{h}^{(k)})$ (the marginal distribution over the top two layers) is an RBM.



1. A DBN with k hidden layers has k weight matrices $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(k)}$.
2. It contains $k + 1$ bias vectors $\mathbf{b}^{(0)}, \dots, \mathbf{b}^{(1)}$, where $\mathbf{b}^{(0)}$ is bias vector for visible layer.
3. Probability distribution represented by DBN is

$$p(\mathbf{h}^{(k-1)}, \mathbf{h}^{(k)}) \propto \exp \left[\mathbf{b}^{(k)\top} \mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)\top} \mathbf{h}^{(k)} + \mathbf{h}^{(k-1)\top} \mathbf{W}^{(k)} \mathbf{h}^{(k)} \right]$$

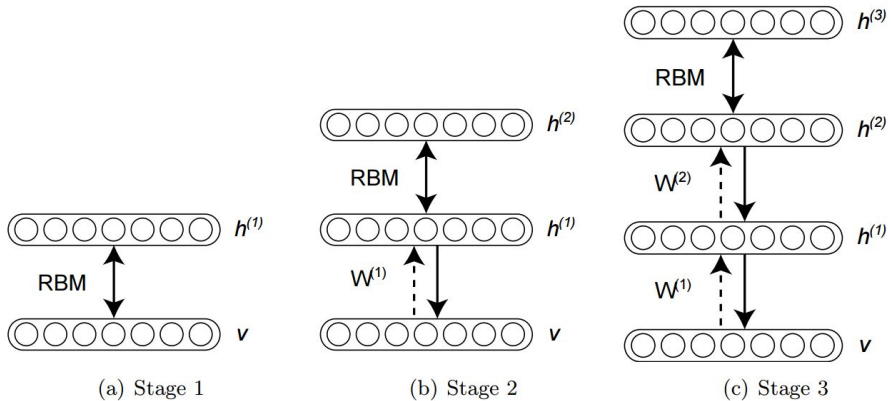
$$p(h_i^{(j)} = 1 | \mathbf{h}^{(j+1)}) = \sigma \left(b_i^{(j)} + \mathbf{W}_{:i}^{(j+1)} \mathbf{h}^{(j+1)} \right)$$

$$p(v_i = 1 | \mathbf{h}^{(1)}) = \sigma \left(b_i^{(0)} + \mathbf{W}_{:i}^{(1)} \mathbf{h}^{(1)} \right)$$

4. For generating a sample from a DBN, do
 - ▶ Use several Gibbs sampling steps from top two hidden layers.
 - ▶ Use a single pass of ancestral sampling through rest of model.

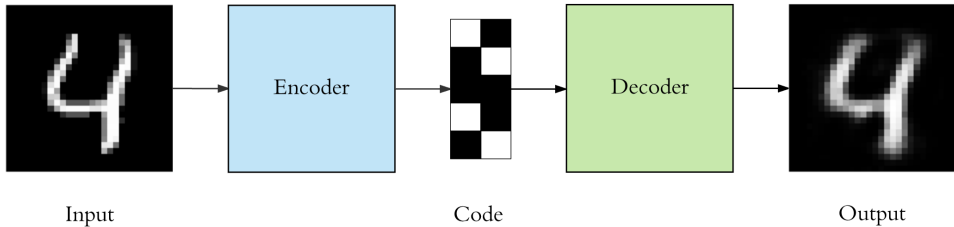


1. Deep belief networks training



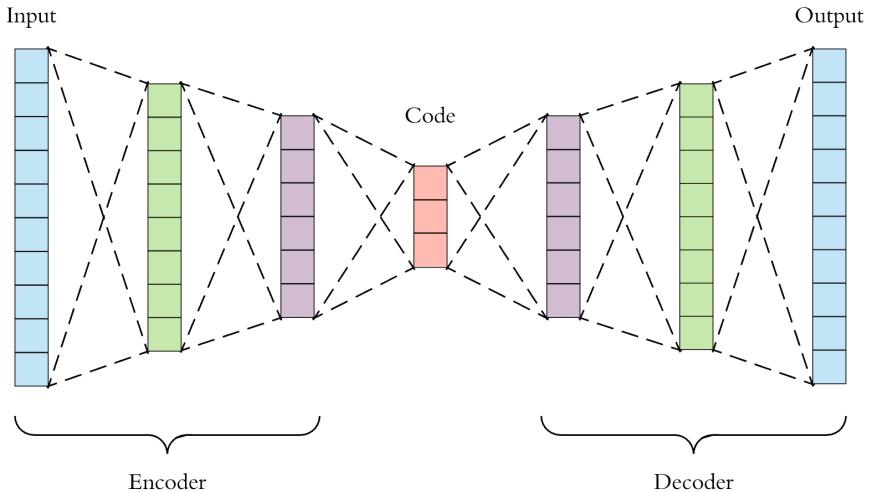
Autoencoder models

1. An autoencoder consists of 3 components: **encoder**, **code** and **decoder**.



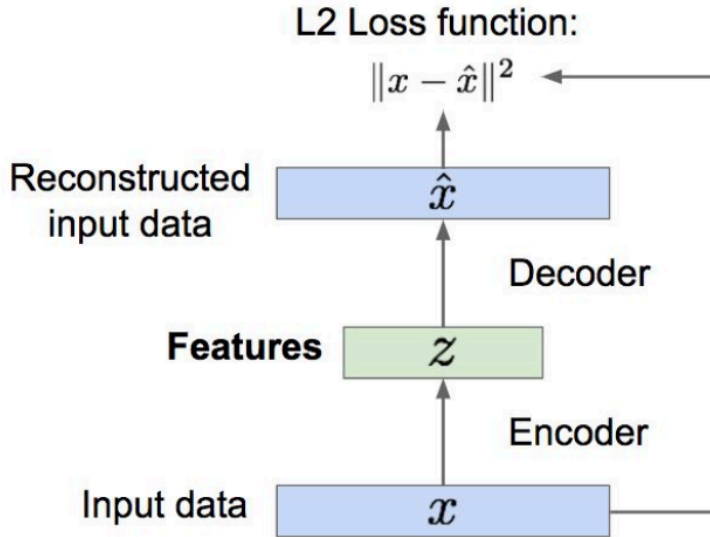
2. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.

1. Autoencoders are simple neural networks that their output is their input.
2. Their goal is to learn how to reconstruct the input-data.



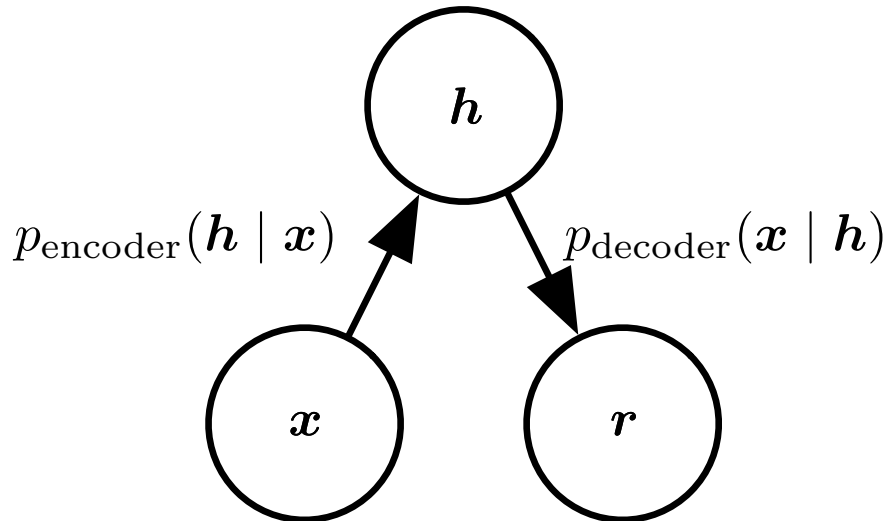


1. We don't use labels but the Autoencoder is trained in supervised manner.



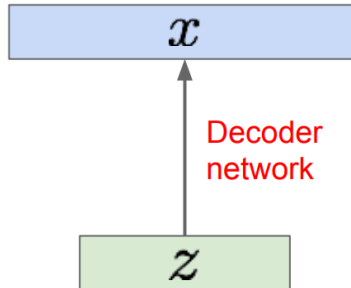


1. The Autoencoder has the following probabilistic model.



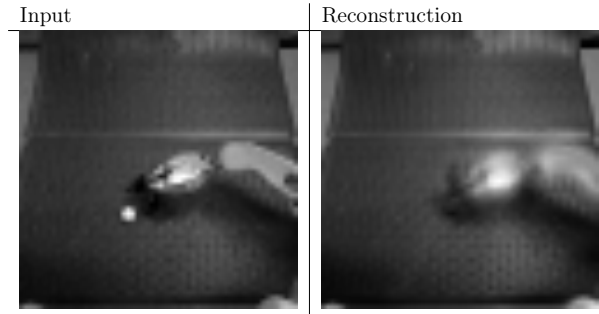


1. Can we generate new sample from an auto encoder?
2. Suppose training data is generated from latent representation z .
3. x is an input sample, z is latent factors used to generate x .



4. How generate a new sample?
 - ▶ Sample from some prior $p(z)$.
 - ▶ Obtain $p(x|z)$.

1. A sample generated from an Autoencoder.



2. MSE can ignore small but task-relevant features.
3. The ping pong ball vanishes because it is not large enough to significantly affect the MSE.
4. Unfortunately, the autoencoder has limited capacity, and the training with MSE did not identify the relevant features.
5. We want to sample from **complex, high-dimensional training distribution**. No direct way to do this! **How do it?**

Autoregressive models

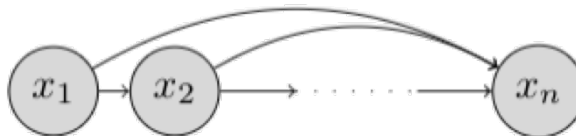


1. We assume we are given access to a dataset $S = \{x_1, x_2, \dots, x_m\}$ of **n-dimensional points** x .
2. For simplicity, we assume points are **binary**, i.e., $x \in \{0, 1\}^n$.
3. Using chain rule, we can factorize the joint distribution as

$$p(x) = p(x_1, x_2, \dots, x_n) = \prod_{i=1}^n p(x_i | x_1, x_2, \dots, x_{i-1}) = \prod_{i=1}^n p(x_i | \mathbf{x}_{<i})$$

where $\mathbf{x}_{<i} = [x_1, x_2, \dots, x_{i-1}]$ denotes the vector of random variables with index less than i .

4. The chain rule factorization can be expressed graphically as a Bayesian network.





1. The autoregressive constraint is a way to model sequential data.
2. The factorization contains n factors and some of these factors contain many parameters ($O(2^n)$ in total).
3. It is infeasible to learn such an exponential number of parameters.
4. AR models use (deep) neural network to parameterize these factors $p(x_i|x_{<i})$.
5. We assume the conditional distributions $p(x_i|x_{<i})$ correspond to Bernoulli random variables and learn a function that maps the preceding random variables x_1, x_2, \dots, x_{i-1} to the mean of this distribution as

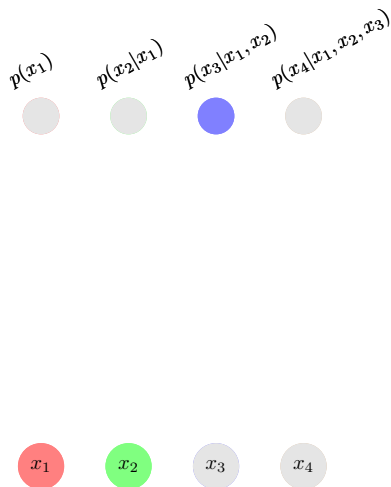
$$p_{\theta_i}(x_i|x_{<i}) = \text{Bern}(f_i(x_1, x_2, \dots, x_{i-1}))$$

where θ_i denotes the set of parameters used to specify the mean function $f_i : \{0, 1\}^{i-1} \mapsto [0, 1]$.

6. The number of parameters of an autoregressive generative model equals to $\sum_{i=1}^n |\theta_i|$.
7. Tractable exact likelihood computations.
8. No complex integral over latent variables in likelihood
9. Slow sequential sampling process.
10. Cannot rely on latent variables.



1. The n th output should only be connected to the previous $n - 1$ inputs.
2. For example, when computing $p(x_4|x_3, x_2, x_1)$ the only inputs that we should consider are x_1, x_2, x_3 because these are the only variables given to us while computing the conditional probability.



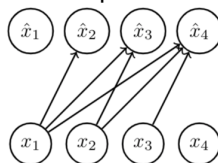


1. In the simplest case, we can specify the function as a linear combination of the input elements followed by a sigmoid non-linearity (to restrict the output to lie between 0 and 1).
2. This gives us the formulation of a **fully-visible sigmoid belief network** (FVSBN).

$$f_i(x_1, x_2, \dots, x_{i-1}) = \sigma \left(a_0^i + \sum_{j=1}^{i-1} a_j^i x_j \right)$$

where σ is sigmoid function and $\theta_i = \{a_0^i, \dots, a_{i-1}^i\}$.

3. At the output layer we want to predict n conditional probability distributions.
4. At the input layer we are given the n input variables.

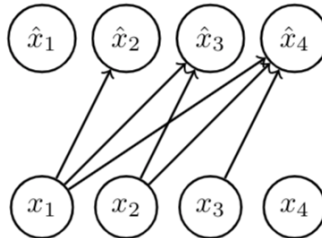


5. The conditional variables $x_i | x_1, \dots, x_{i-1}$ are Bernoulli with parameters

$$\hat{x}_i = p(x_i = 1 | x_1, \dots, x_{i-1}; \theta_i) = \sigma \left(a_0^i + \sum_{j=1}^{i-1} a_j^i x_j \right)$$

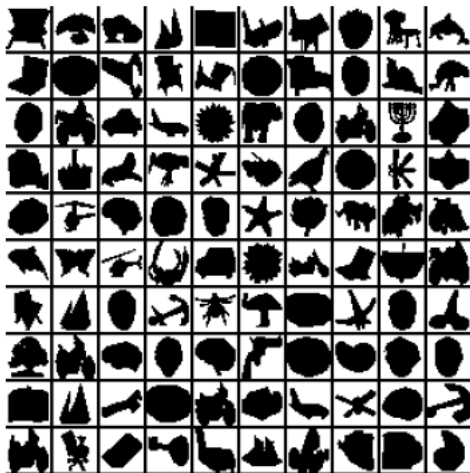


1. How to evaluate $p(x_1, \dots, x_{900})$?
2. Multiply all the **conditionals factors**.
3. How to sample from $p(x_1, \dots, x_{900})$?
 - ▶ Sample $\bar{x}_1 \sim p(x_1)$.
 - ▶ Sample $\bar{x}_2 \sim p(x_2 | x_1 = \bar{x}_1)$.
 - ▶ Sample $\bar{x}_3 \sim p(x_3 | x_1 = \bar{x}_1, x_2 = \bar{x}_2)$.
4. How many parameters? $1 + 2 + 3 + \dots + n \approx \frac{n^2}{2}$

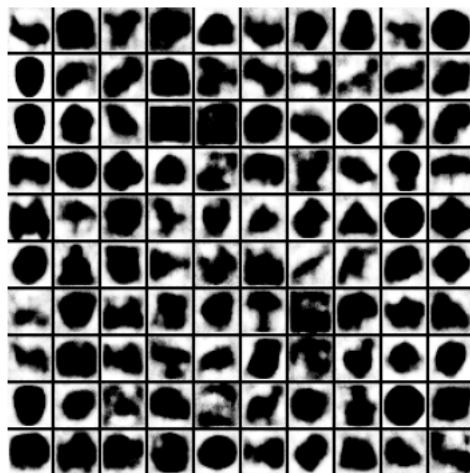


5. This model is called **Fully Visible Sigmoid Belief Network (FVSBN)**.

1. **Left:** Training (Caltech 101 Silhouettes)



Right: Samples from the model



Autoregressive models

Neural Autoregressive Density Estimator



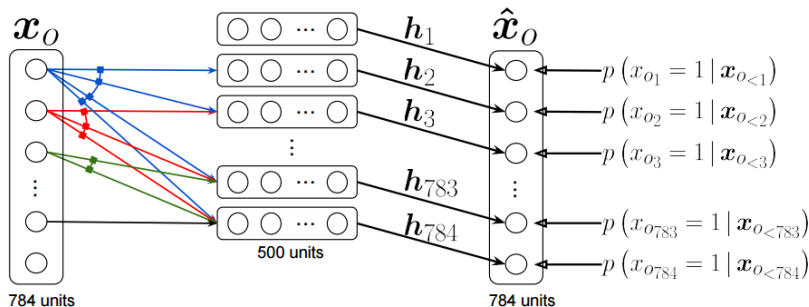
1. To increase the expressiveness of an autoregressive generative model, we can use more flexible parameterizations for the mean function such as MLP instead of logistic regression.
2. For example, consider the case of a neural network with one hidden layer.
3. The mean function for variable i can be expressed as

$$\mathbf{h}_i = \sigma(A_i \mathbf{x}_{<i} + \mathbf{c}_i)$$

$$f_i(x_1, x_2, \dots, x_{i-1}) = \sigma(\mathbf{a}'\mathbf{h}_i + b_i)$$

where $\mathbf{h}_i \in \mathbb{R}^d$ is hidden layer activations of MLP.

4. Hence, we have the following architecture





1. To improve model, use a neural network with one hidden layer instead of logistic regression.

$$\begin{aligned}\mathbf{h}_i &= \sigma(A_i \mathbf{x}_{<i} + \mathbf{c}_i) \\ \hat{x}_i &= p(x_i = 1 | x_1, \dots, x_{i-1}; \boldsymbol{\theta}^i) = \sigma(\boldsymbol{\alpha}^{(i)} \mathbf{h}_i + b_i) \\ \boldsymbol{\theta}^i &= \{A_i, \mathbf{c}_i, \boldsymbol{\alpha}^{(i)}, b_i\}\end{aligned}$$

2. $\mathbf{h}_i \in \mathbb{R}^d$ denotes the hidden layer activations for the MLP.
3. $\theta_i = \{A_i \in \mathbb{R}^{d \times (i-1)}, \mathbf{c}_i \in \mathbb{R}^d, \boldsymbol{\alpha}^{(i)} \in \mathbb{R}^d, b_i \in \mathbb{R}\}$ are the set of parameters.
4. Hidden layer parameters are shared and **only the relevant columns of \mathbf{A} are used for each i .**
5. The total number of parameters in this model is dominated by the matrices A_i and given by $O(nd + n)$.



1. The **Neural Autoregressive Density Estimator (NADE)** provides an alternate MLP-based parameterization that is more statistically and computationally efficient than the given approach (Larochelle and Murray 2011).
2. In NADE, parameters are shared across the functions used for evaluating the conditionals.
3. The hidden layer activations are specified as

$$\mathbf{h}_i = \sigma(W_{\cdot, < i} \mathbf{x}_{< i} + \mathbf{c})$$
$$\hat{x}_i = p(x_i = 1 | x_1, \dots, x_{i-1}; \boldsymbol{\theta}^i) = \sigma(\boldsymbol{\alpha}^{(i)} \mathbf{h}_i + b_i)$$

4. $\theta = \{W \in \mathbb{R}^{d \times n}, \mathbf{c} \in \mathbb{R}^d, \{\boldsymbol{\alpha}^{(i)} \in \mathbb{R}^d\}_{i=1}^n, \{b_i \in \mathbb{R}\}_{i=1}^n\}$ is the full set of parameters.
5. The weight matrix W and the bias vector c are shared across the conditionals.



1. Sharing parameters has two benefits:
 - ▶ The total number of parameters gets reduced from $O(n^2d)$ to $O(nd)$.
 - ▶ The hidden unit activations can be evaluated in $O(nd)$ time via the following recursive strategy:

$$\mathbf{h}_i = \sigma(\mathbf{a}_i)$$
$$\mathbf{a}_{i+1} = \mathbf{a}_i + W[:, i]x_i$$

with the base case given by $\mathbf{a}_1 = \mathbf{c}$.

2. Training of NADE is done by minimizing the average negative log-likelihood of the parameters given the training set:

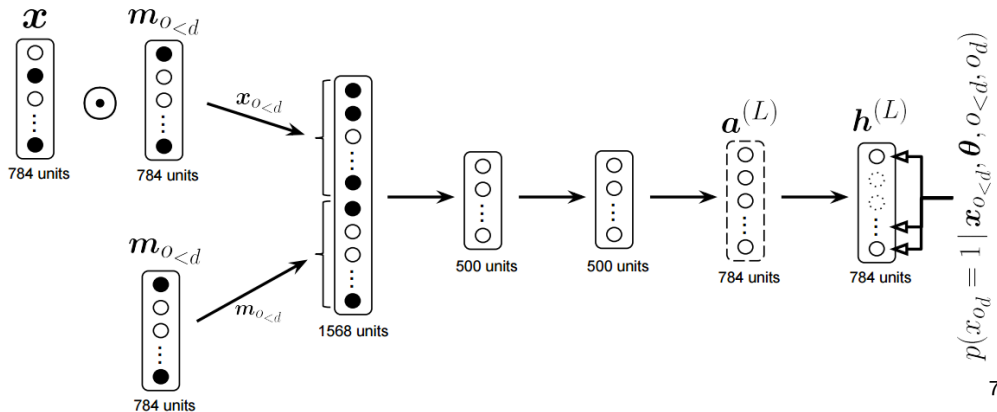
$$-\frac{1}{T} \sum_{i=1}^T \log p(x_i)$$



1. Samples from NADE trained on a binary version of MNIST.



1. The input to the network (DeepNADE) is the concatenation of the masked data and the mask itself (Uria, Côté, et al. 2016).
2. This allows the network to identify cases when input data is truly zero from cases when input data is zero because of the mask.
3. NADE also explored other autoencoder architectures such as convolutional neural networks
4. DeepNade with two hidden layers





1. The RNADE algorithm extends NADE to learn generative models over real-valued data (Uria, Murray, and Larochelle 2013).
2. For real-valued variables, the conditionals are modeled via a continuous distribution such as **mixture of K Gaussian**.
3. Instead of learning a mean function, we now learn the means $\mu_{i,1}, \mu_{i,2}, \dots, \mu_{i,K}$, variances $\sigma_{i,1}, \sigma_{i,2}, \dots, \sigma_{i,K}$, and probability of sampling from each mixture $\pi_{i,1}, \pi_{i,2}, \dots, \pi_{i,K}$ of the K Gaussian for every conditional.

$$p(x_i | x_{<i}) = \sum_{j=1}^K \pi_{ij} \mathcal{N}(\mu_{ij}, \sigma_{ij}^2)$$

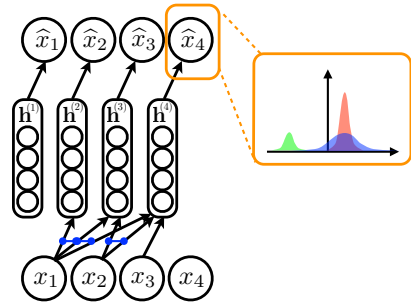
1. Output of the network are parameters of a mixture model for $p(x_k|x_{<k})$
2. Means are $\mu_{i,k} = b_{i,k}^{\mu_i} + \alpha_{i,k}^{\mu_i} h_i$
3. Standard deviations are

$$\sigma_{i,k} = \exp \left(b_{i,k}^{\sigma_i} + \alpha_{i,k}^{\sigma_i} h_i \right)$$

4. Mixing weights are

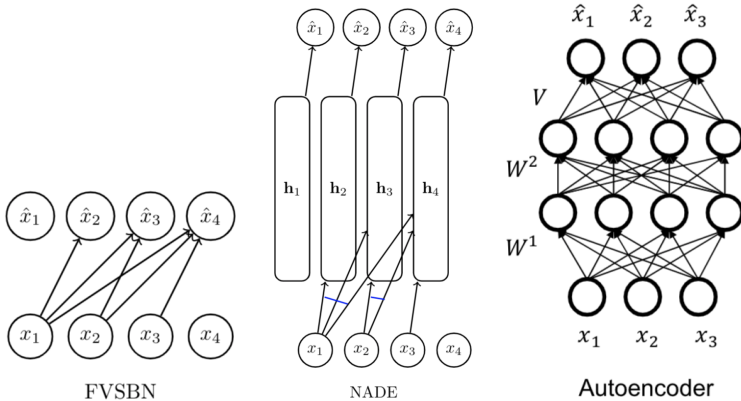
$$\pi_{i,k} = \text{softmax} \left(b_{i,k}^{\pi_i} + \alpha_{i,k}^{\pi_i} h_i \right)$$

5. Please study **DocNADE**.





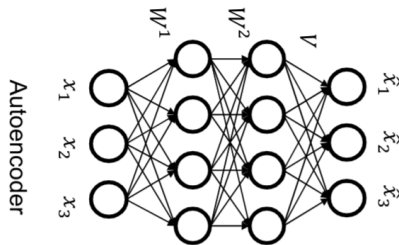
1. Considering the following models.



2. FVSN and NADE look similar to an autoencoder.
3. An encoder computing hidden.
4. A decoder computing densities.
5. A loss function, which is likelihood.



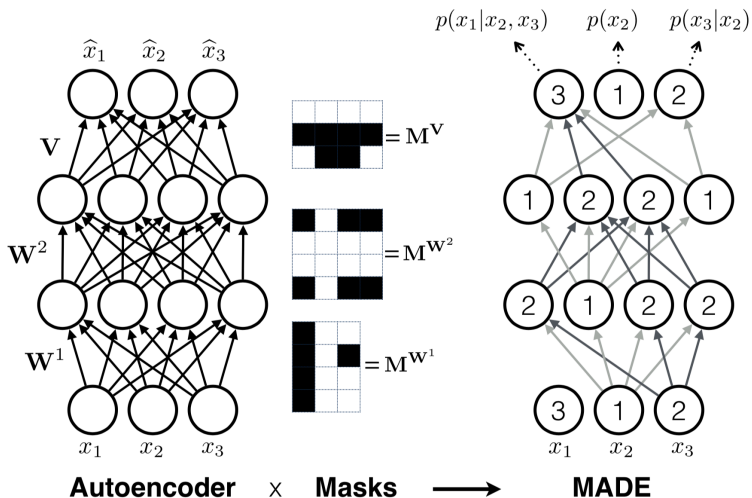
1. An autoencoder is not a generative model: it does not define a distribution over x for sampling new data points.



2. Can we get a generative model from an autoencoder?
3. We need to make sure it corresponds to a valid Bayesian Network, i.e., we need an ordering. If the ordering is 1, 2, 3, then
 - ▶ \hat{x}_1 cannot depend on any input x .
 - ▶ \hat{x}_2 can only depend on x_1 .
4. We can use a single neural network to produce all the parameters.



- MADE is an autoencoder that preserves autoregressive property (Germain et al. 2015).





1. MADE is a specially designed architecture to enforce the autoregressive property in the autoencoder efficiently.
2. MADE removes the contribution of certain hidden units by using mask matrices so that each input dimension is reconstructed only from previous dimensions in a given ordering in a single pass.
3. In a multilayer fully-connected neural network, say, we have L hidden layers with weight matrices $\mathbf{W}^1, \dots, \mathbf{W}^L$ and an output layer with weight matrix \mathbf{V} . The output $\hat{\mathbf{x}}$ has dimensions $\hat{x}_i = p(x_i | x_{1:i-1})$
4. Without any mask, we have

$$\mathbf{h}^0 = \mathbf{x}$$

$$\mathbf{h}^l = \text{activation}'(\mathbf{W}^l \mathbf{h}^{l-1} + \mathbf{b}^l)$$

$$\hat{\mathbf{x}} = \sigma(\mathbf{V} \mathbf{h}^L + \mathbf{c})$$



1. Without any mask, we have

$$\mathbf{h}^0 = \mathbf{x}$$

$$\mathbf{h}^l = \text{activation}^l(\mathbf{W}^l \mathbf{h}^{l-1} + \mathbf{b}^l)$$

$$\hat{\mathbf{x}} = \sigma(\mathbf{V} \mathbf{h}^L + \mathbf{c})$$

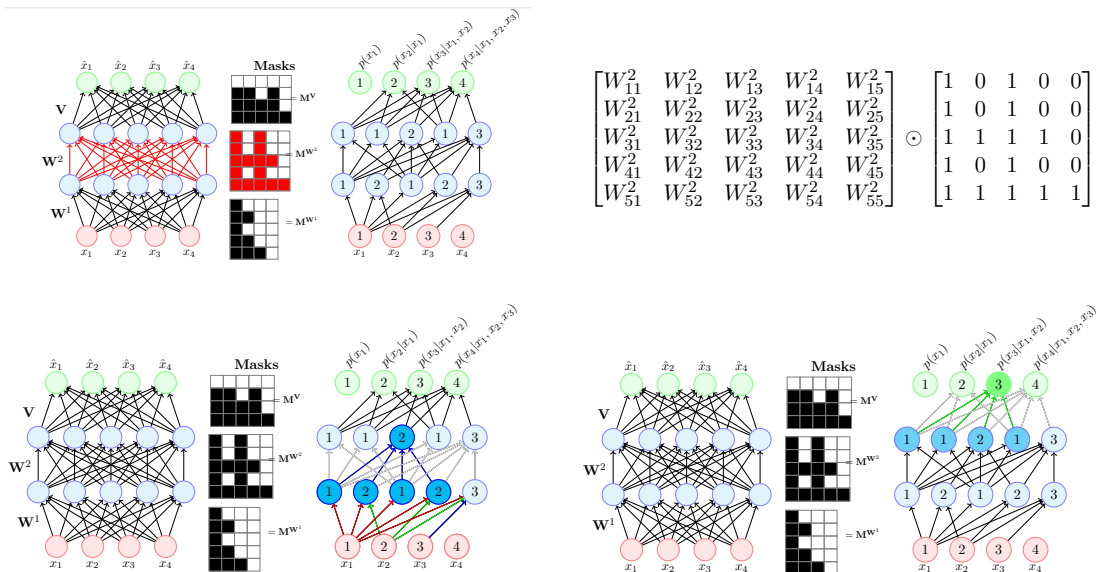
2. To zero out some connections between layers, we can simply element-wise multiply every weight matrix by a binary mask matrix.

$$\mathbf{h}^l = \text{activation}^l((\mathbf{W}^l \odot \mathbf{M}^{\mathbf{W}^l}) \mathbf{h}^{l-1} + \mathbf{b}^l)$$

$$\hat{\mathbf{x}} = \sigma((\mathbf{V} \odot \mathbf{M}^{\mathbf{V}}) \mathbf{h}^L + \mathbf{c})$$

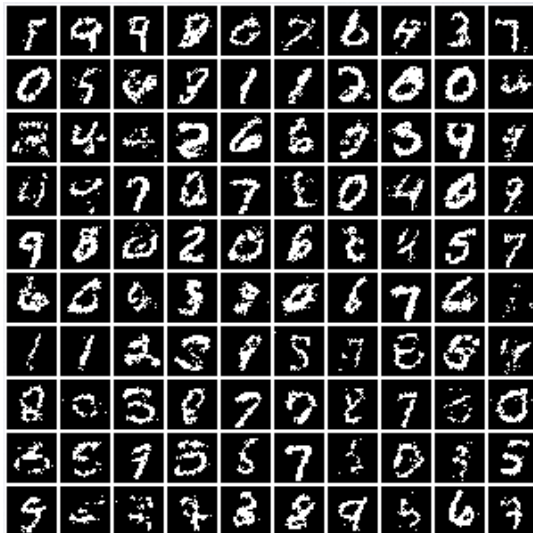
3. Mask matrix is constructed by a labeling process.

Masked Autoencoder for Distribution Estimation (MADE)





1. The results of MADE on MNIST.





1. We know the structure (Markov random field) of the data (Khajenezhad, Madani, and Beigy 2021).
2. In structured distributions, the graph structure of the variables declares their conditional dependencies.
3. Therefore, having a graph structure, each of the chain rule conditional terms might be presentable by a conditional probability on a smaller set of variables.
4. In other words, for each i , we can assume that there is a subset $B_i \subseteq \{1, \dots, i-1\}$ such that $p(x_i|x_{<i}) = p(x_i|x_{B_i})$.
5. We call B_i as **Looking-back Markov blanket of the i -th dimension**. Then

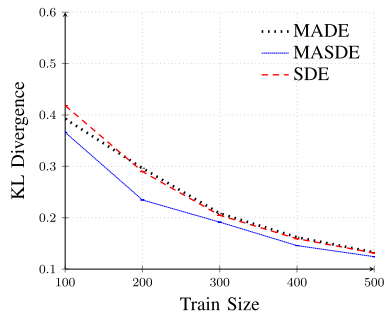
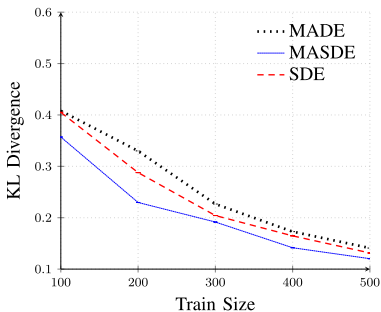
$$p(x_1, \dots, x_d) = P(x_1)p(x_2|x_{B_2}) \dots P(x_d|x_{B_d})$$

6. Use an autoencoder that has the above autoregressive property.
7. Mask matrix is constructed by a labeling process.



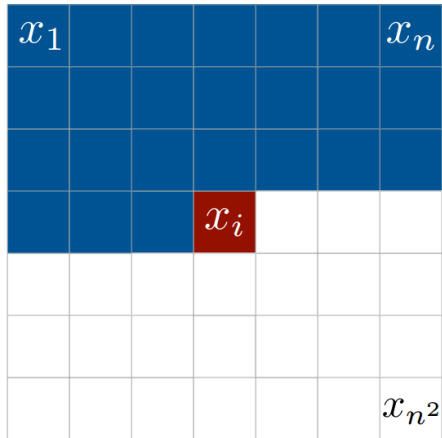
- MASDE needs a smaller training set in comparison with its counterparts.

Size of the
Hidden Layers = 100

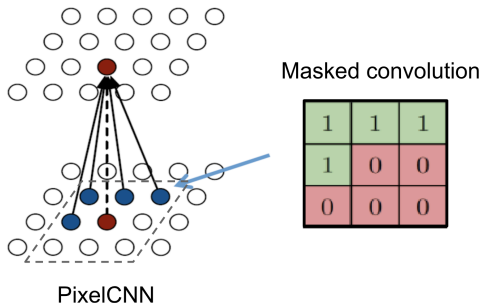




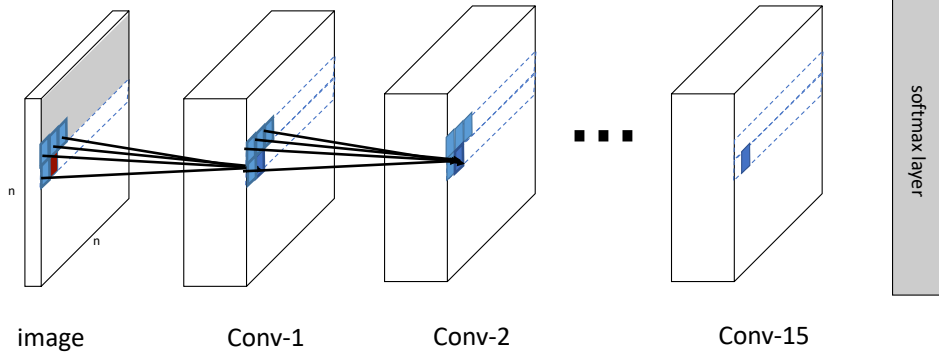
1. PixelRNN is a deep generative model for images (Oord, Kalchbrenner, and Kavukcuoglu 2016).
2. Dependency on previous pixels modeled using an RNN (LSTM).



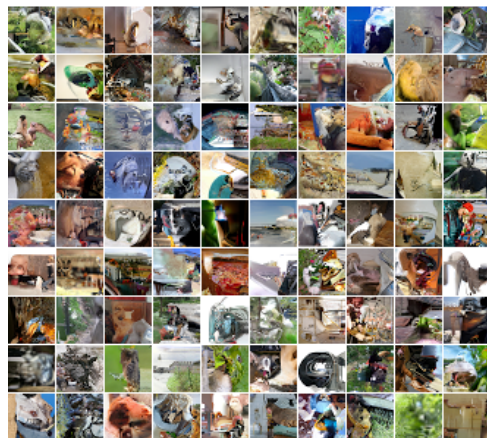
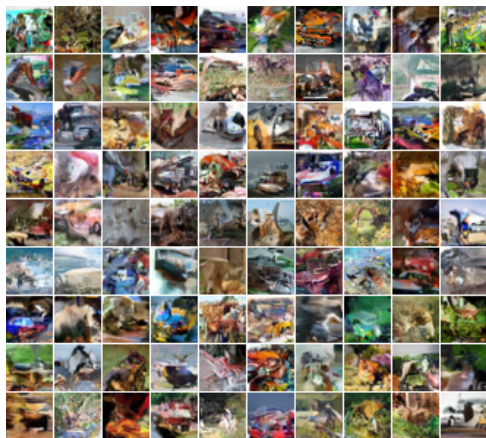
1. The main drawback of PixelRNN is that training is very slow.
2. PixelCNN uses standard convolutional layers to capture a bounded receptive field and compute features for all pixel positions at once (Oord, Kalchbrenner, Espeholt, et al. [2016](#)).
3. In PixelCNN, pooling layers are not used.
4. Masks are adopted in the convolutions to restrict the model from violating the conditional dependence.



5. Please also PixelCNN++ (Salimans, Karpathy, et al. [2017](#)).

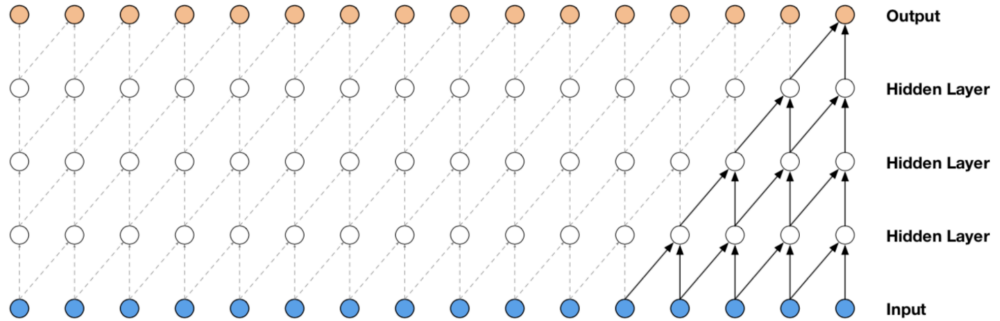


1. The training set (CIFAR-10 (left)) and the samples generated by the PixelCNN (right).

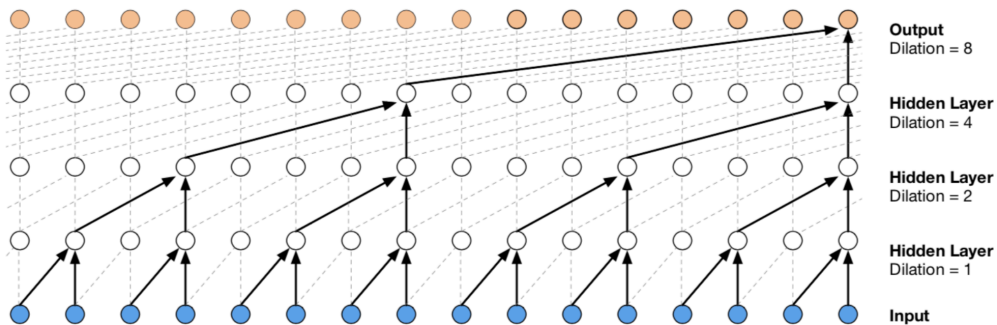




1. WaveNet is very similar to PixelCNN but applied to 1-D audio signals (Oord, Dieleman, et al. 2016).
2. WaveNet consists of a stack of **causal convolution** which is a convolution operation designed to respect the ordering.
3. Causal convolutions are a type of convolution used for temporal data which ensures the model cannot violate the ordering in which we model the data: the prediction $p(x_{t+1}|x_1, \dots, x_t)$.
4. The causal convolution in WaveNet is simply to shift the output by a number of timestamps to the future so that the output is aligned with the last input element.



1. One big drawback of convolution layer is a very limited size of receptive field.
2. WaveNet therefore adopts **dilated convolution**, where the kernel is applied to an evenly-distributed subset of samples in a much larger receptive field of the input.

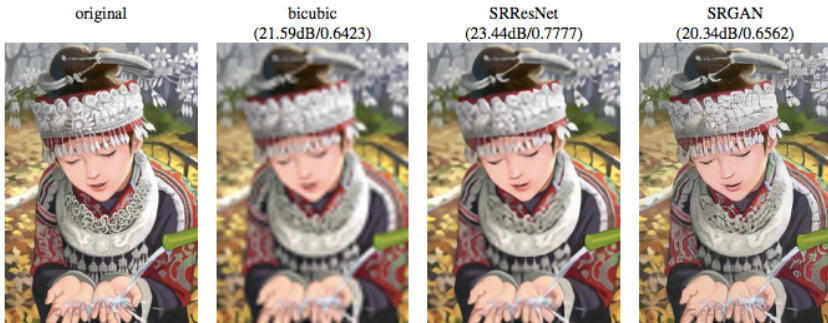


Generative Adversarial Networks



1. Generative adversarial networks (GANs) are relatively new (I. J. Goodfellow et al. 2014).
2. GANs are a new way to build generative models $P(x)$.
3. Generative adversarial networks
 - ▶ **Generative:** Learns a generative model.
 - ▶ **Adversarial:** Trained in an adversarial setting
 - ▶ **Networks:** Use Deep Neural Networks

1. Which one is Computer generated?



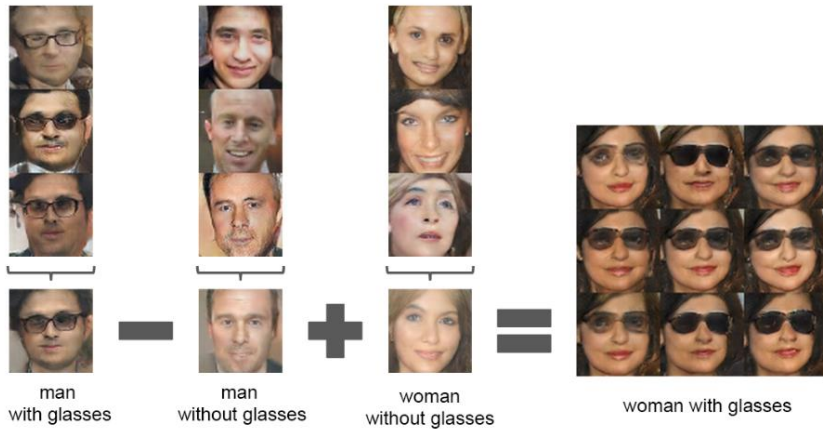
2. How do we generate a fake image?

3. Can we generate a fake image from a random number?

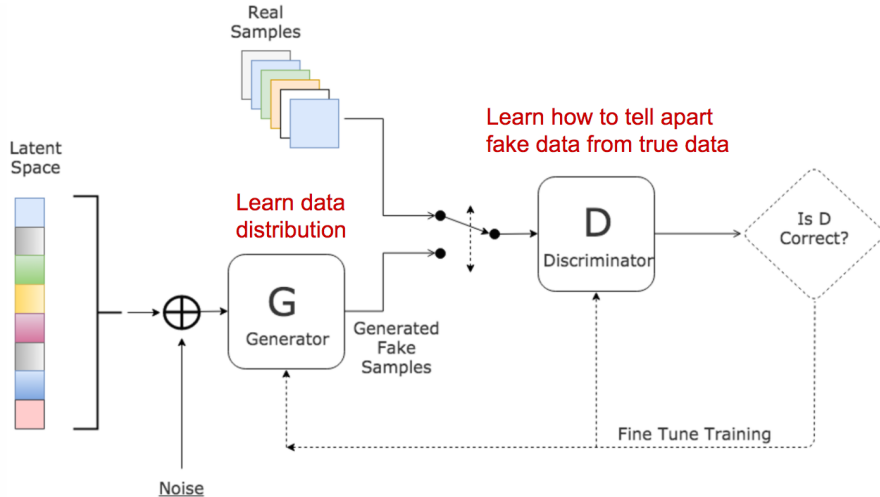
1. Results obtained from GAN (Radford, Metz, and Chintala 2016).



1. Results obtained from GAN (Radford, Metz, and Chintala [2016](#)).

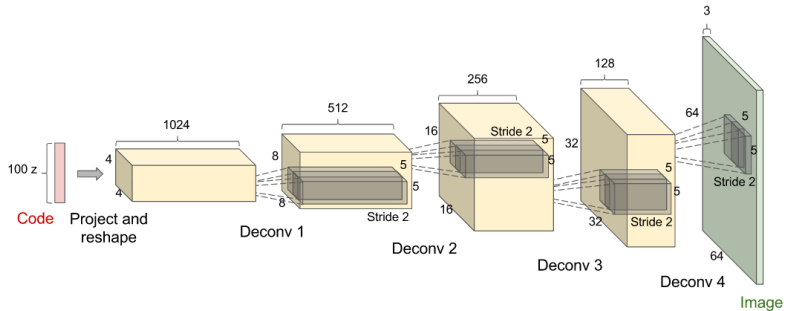


1. GAN has the following architecture

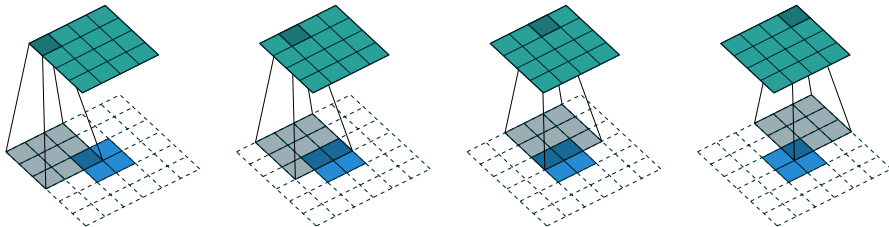


2. Z (input to generator) is some random noise (Gaussian/Uniform).
3. Z can be thought as the latent representation of the image.

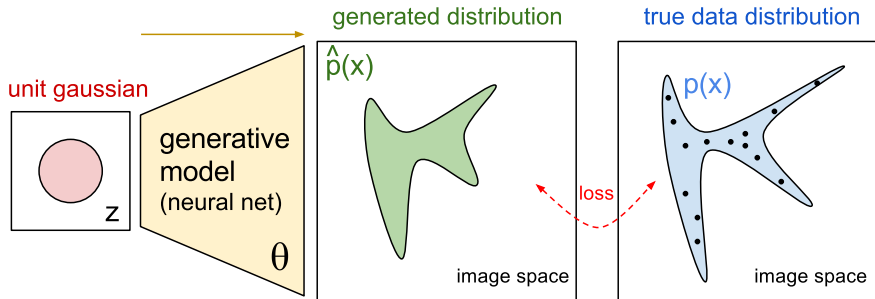
1. Opposite of convolutional neural nets.



2. Deconvolution layer or transposed convolutional layer is pad the original input (blue entries) with zeroes (white entries) (Dumoulin and Visin 2016).



1. How to train it?





1. The generator tries to learn $P(x|z)$.
2. Inputs are directly sampled from $Q(z)$.
3. **Problem:** No true data x is provided when training the generator
4. Instead of a traditional loss function, gradient is provided by a discriminator (another network)



1. The discriminator attempts to tell the difference between real and fake images.
2. It tries to learn $P(y|x)$, where y is the label (real or generated) and x is the real or generated data.
3. Trained using standard **cross entropy loss** to assign the correct label (although this has changed in recent GANs).
4. Generator weights are frozen while training discriminator; inputs are generated data and real data, targets are 0 and 1
5. From generator's point-of-view, discriminator is a black-box loss function



1. Let x be a sample (fake or real).
2. Let $D(x)$ be the probability that x came from real data rather than p_g .
3. For a fake sample $G(z)$, the discriminator is expected to output a probability, $D(G(z))$, close to zero by maximizing $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$.
4. For real data, the discriminator is expected to output a probability x , close to one by maximizing $\mathbb{E}_{x \sim p_{data}(x)}[\log D(x)]$.
5. The generator is trained to increase the chances of D producing a high probability for a fake example, thus to minimize $\mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$.
6. When combining both aspects together, D and G are playing a minimax game in which we should optimize the following loss function:

$$\begin{aligned} \min_G \max_D V(D, G) &= \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \\ &= \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{x \sim p_g(x)}[\log(1 - D(x))] \end{aligned}$$

7. $\mathbb{E}_{x \sim p_{data}(x)}[\log D(x)]$ has no impact on G during gradient descent updates.



1. Loss function is

$$\begin{aligned} V(G, D) &= \int_x p_r(x) \log(D(x)) dx + \int_z p_z(z) \log(1 - D(G(z))) dz \\ &= \int_x \left(p_r(x) \log(D(x)) + p_g(x) \log(1 - D(x)) \right) dx \end{aligned}$$

2. The full two-player game can be summarily described by the below.

$$\min_G \max_D V(D, G)$$



1. It is important to understand that both the generator and discriminator are trying to learn **moving targets**. **Both networks are trained simultaneously**.
2. The discriminator needs to update based on how well the generator is doing.
3. The generator is constantly updating to improve performance on the discriminator.
4. These two need to be balanced correctly to achieve stable learning instead of chaos.



Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

Discriminator
updates

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

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

end for

Generator
updates

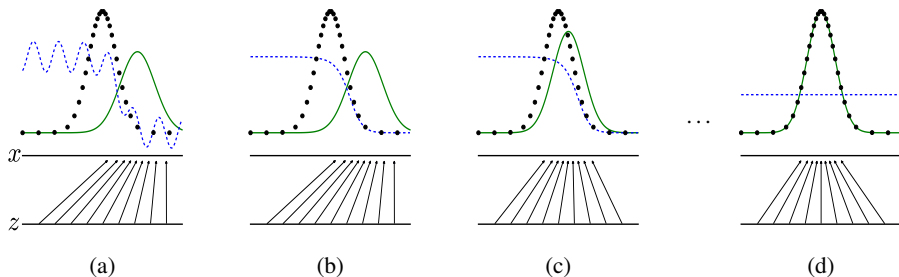
- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by descending its stochastic gradient:

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

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

1. How GAN is trained?



2. discriminative distribution $D(x)$, real data p_{data} , generative distribution p_g .

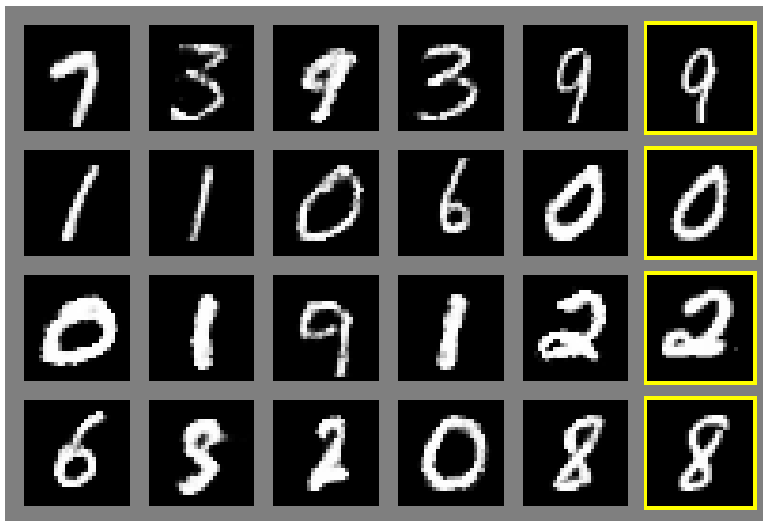
(a) An adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier.

(b) In inner loop of algorithm, D is trained to discriminate samples from data, converging to $D^*(x)$.

(c) After an update to G , gradient of D has guided $G(z)$ to flow to regions that are more likely to be classified as data.

(d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$.

1. Visualization of samples from the model.
2. Rightmost column shows the nearest training example of the neighboring sample.



1. Digits obtained by linearly interpolating between coordinates in z space of the full model.





Theorem (Optimality of GAN)

For G fixed, the optimal discriminator D is

$$D^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

Theorem (Convergence of training algorithm of GAN)

If G and D have enough capacity, and at each step of training Algorithm, the discriminator is allowed to reach its optimum given G , and p_g is updated so as to improve the criterion $V(D, G)$, then, p_g converges to p_{data}



1. When both G and D are at their optimal values, we have $p_g = p_{data}$ and $D^*(x) = \frac{1}{2}$, and the loss function becomes:

$$\begin{aligned} V(G, D^*) &= \int_x \left(p_{data}(x) \log(D^*(x)) + p_g(x) \log(1 - D^*(x)) \right) dx \\ &= \log \frac{1}{2} \int_x p_{data}(x) dx + \log \frac{1}{2} \int_x p_g(x) dx \\ &= -2 \log 2 \end{aligned}$$



1. KL divergence measures how one probability distribution p diverges from a second probability distribution q

$$D_{KL}(p||q) = \int_x p(x) \log \frac{p(x)}{q(x)} dx$$

2. KL divergence is asymmetric.
3. In cases where $p(x)$ is close to zero, but $q(x)$ is significantly non-zero, the q 's effect is disregarded.
4. Jensen–Shannon Divergence is a measure of similarity between two probability distributions, bounded by $[0, 1]$.

$$D_{JS}(p||q) = \frac{1}{2} D_{KL} \left(p \parallel \frac{p+q}{2} \right) + \frac{1}{2} D_{KL} \left(q \parallel \frac{p+q}{2} \right)$$

5. JS divergence is symmetric and more smooth.



1. JS divergence between p_{data} and p_g can be computed as:

$$\begin{aligned} D_{JS}(p_{data}||p_g) &= \frac{1}{2} D_{KL} \left(p_{data} || \frac{p_{data} + p_g}{2} \right) + \frac{1}{2} D_{KL} \left(p_g || \frac{p_{data} + p_g}{2} \right) \\ &= \frac{1}{2} \left(\log 2 + \int_x p_{data}(x) \log \frac{p_{data}(x)}{p_{data} + p_g(x)} dx \right) \\ &\quad + \frac{1}{2} \left(\log 2 + \int_x p_g(x) \log \frac{p_g(x)}{p_{data} + p_g(x)} dx \right) \\ &= \frac{1}{2} \left(\log 4 + V(G, D^*) \right) \end{aligned}$$

2. Thus

$$V(G, D^*) = 2D_{JS}(p_{data}|p_g) - 2\log 2$$

3. The best G^* that replicates the real data distribution leads to the minimum $V(G^*, D^*) = -2\log 2$, which is aligned to the optimal solution.



1. Hard to achieve Nash equilibrium (Salimans, I. J. Goodfellow, et al. 2016).
2. Low dimensional supports: **When the intrinsic dimension is low, then training GAN will be unstable** (Arjovsky and Leon Bottou 2017).
3. Vanishing gradient: **When the discriminator is perfect, loss function is zero and there is not any training.**
4. Mode collapse: **During the training, the generator may collapse to a setting where it always produces same outputs.**
5. Lack of a proper evaluation metric



1. **Feature Matching:** This suggests to optimize the discriminator to inspect whether the generator's output matches expected statistics of the real samples. New objective function

$$\|\mathbb{E}_{x \sim p_{data}} f(x) - \mathbb{E}_{z \sim p_z(z)} f(G(z))\|_2^2$$

where $f(x)$ can be any computation of statistics of features, such as mean or median.

2. **Mini-batch Discrimination:** Instead of processing each point independently, the discriminator is able to digest the relationship between training data points in one batch.
3. **Historical Averaging:** This adds a term penalizes the training speed when parameters are changing too dramatically in time.
4. **One-sided Label Smoothing:** When feeding the discriminator, instead of providing 1 and 0 labels, use soften values such as 0.9 and 0.1



5. **Virtual Batch Normalization:** Each data sample is normalized based on a fixed batch (**reference batch**) of data rather than within its minibatch. The **reference batch** is chosen once at the beginning and stays the same through the training.
6. **Adding Noises:**
7. **Use Better Metric of Distribution Similarity:** The JS divergence fails to provide a meaningful value when two distributions are disjoint. **Wasserstein metric** is introduced.

Generative Adversarial Networks

Wasserstein GAN



1. **Wasserstein Distance** is a measure of the distance between two probability distributions.
2. When dealing with the continuous probability domain, the distance becomes

$$W(p_{data}, p_g) = \inf_{\gamma \sim \Pi(p_{data}, p_g)} \mathbb{E}_{(x,y) \sim \gamma} [\|x - y\|]$$

where $\Pi(p_{data}, p_g)$ is the set of all possible joint probability distributions between p_{data} and p_g .

3. It is intractable to exhaust all the possible joint distributions in $\Pi(p_{data}, p_g)$ to compute $\inf_{\gamma \sim \Pi(p_{data}, p_g)}$, the following metric is used.

$$W(p_{data}, p_g) = \frac{1}{K} \sup_{\|f\|_L \leq K} \mathbb{E}_{x \sim p_{data}} [f(x)] - \mathbb{E}_{x \sim p_g} [f(x)]$$

where $\|f\|_L \leq K$ means that f is K -Lipschitz.

4. **what are their meaning and their difference?**



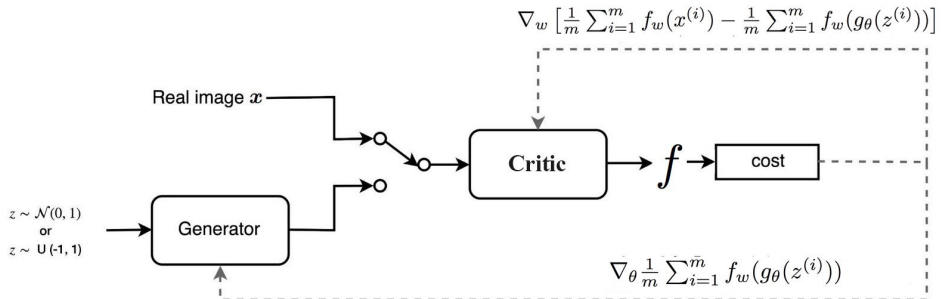
1. In WGAN, discriminator network instead of producing the probability of generating real data, the network produces a scalar score (Arjovsky and Leon Bottou 2017).
2. This score can be interpreted as how real the input images are.
3. In reinforcement learning, we call it the **value function** which measures how good a input is.
4. We rename the discriminator to **critic** to reflect its new role.
5. The loss function for WGAN is

$$V(p_{data}, p_g) = W(p_{data}, p_g) = \max_{w \in W} \mathbb{E}_{x \sim p_{data}} [f_w(x)] - \mathbb{E}_{z \sim p_z(z)} [f_w(g_\theta(z))]$$

f comes from a family of K -Lipschitz continuous functions $\{f_w\}_{w \in W}$ parameterized by w .

6. The **discriminator** model is used for learning w to find a good f_w and the loss function is configured as measuring the Wasserstein distance between p_{data} and p_g .

1. WGAN architecture is (Arjovsky, Chintala, and Léon Bottou [2017](#)).



Generative Adversarial Networks

Conditional GAN



1. In GAN, we have two neural nets: the generator $G(z)$ and the discriminator $D(x)$.
2. Now, as we want to condition those networks with some vector y .
3. The easiest way to do it is to feed y into both networks (Mirza and Osindero 2014).
4. Hence, generator and discriminator are now $G(z, y)$ and $D(x, y)$, respectively.
5. We can see it with a probabilistic point of view. $G(z, y)$ is modeling the distribution of our data, given z and y , that is, the data is generated with this scheme $x_G \sim G(x|z, y)$
6. Likewise for the discriminator, now it tries to find discriminating label for x and x_G , that are modeled with $d \sim D(d|x, y)$.



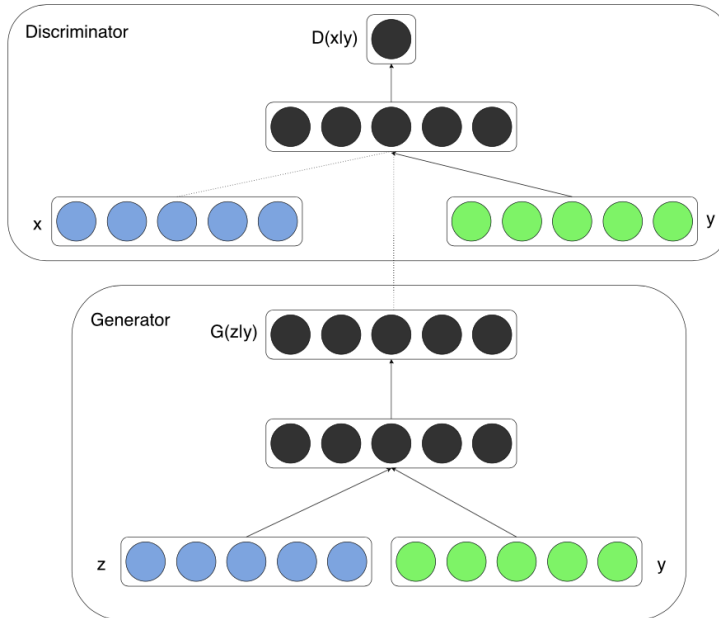
1. Hence, we could see that both D and G is jointly conditioned to two variables z or x and y .
2. Now, the objective function is given by:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x, y)] \\ + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z, y), y))]$$

3. If we compare the above loss to GAN loss, the difference only lies in the additional parameter y in both D and G .



1. The architecture of CGAN is





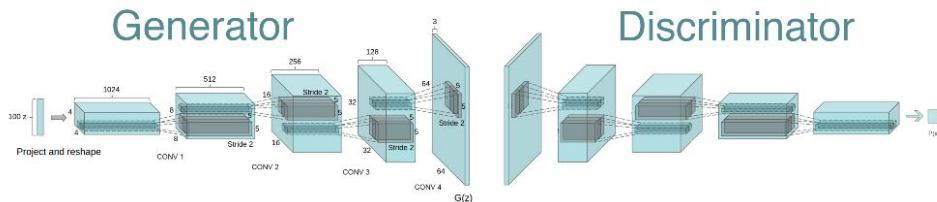
1. The following figure shows some of the generated samples.
2. Each row is conditioned on one label and each column is a different generated sample.



Generative Adversarial Networks

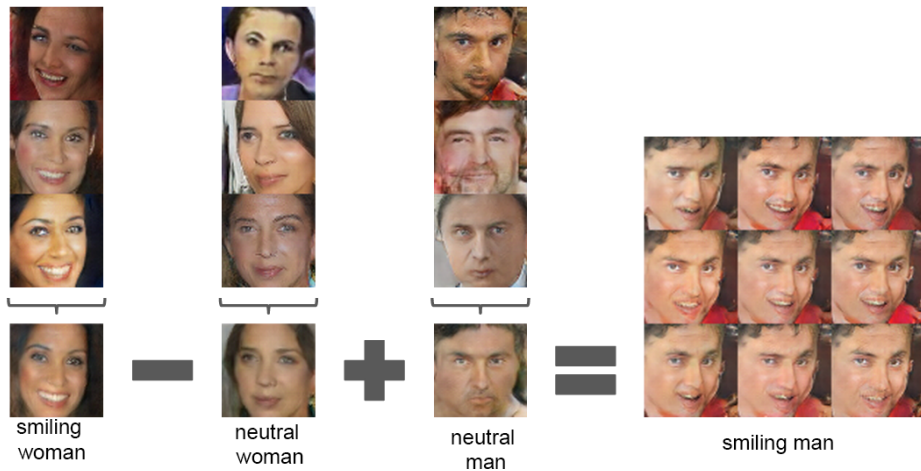
Deep Convolutional GAN

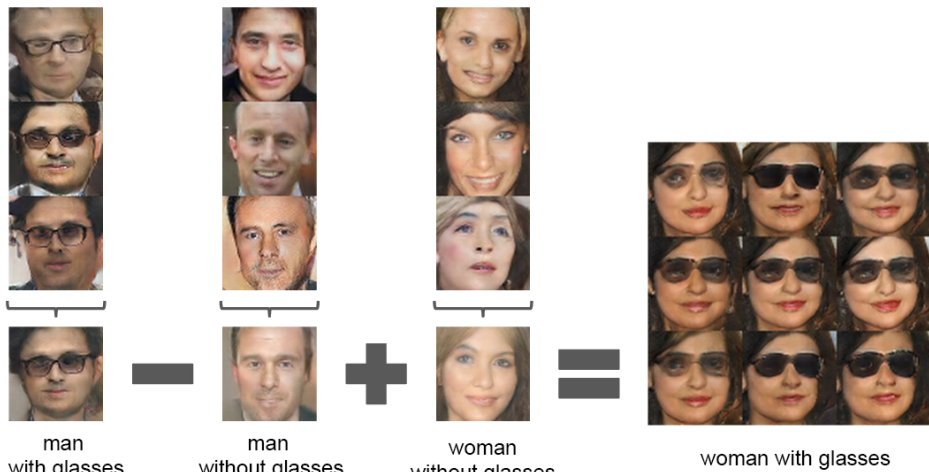
1. DCGAN maps from random noise to an image matrix.
2. It uses convolutional Layers in the generator network to produce better results (Radford, Metz, and Chintala 2016).
3. Combine CNN and GAN for unsupervised learning.
4. Learns a hierarchy of feature representations



5. Replace any pooling layers with strided convolutions.
6. Use batch-normalization in both the generator and the discriminator.
7. Remove fully connected hidden layers for deeper architectures.
8. Use ReLU activation in generator for all layers except for the output, which uses Tanh.
9. Use LeakyReLU activation in the discriminator for all layers.



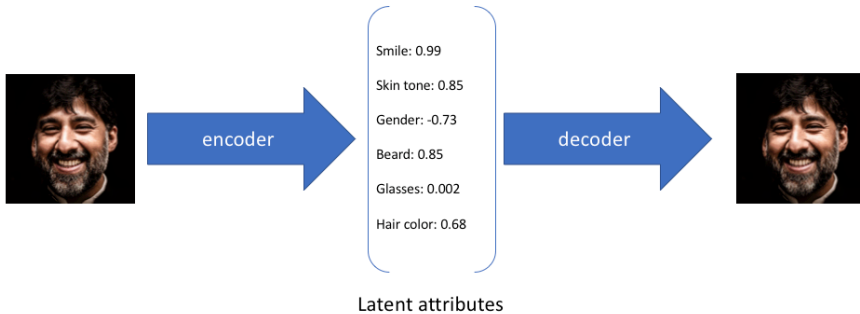




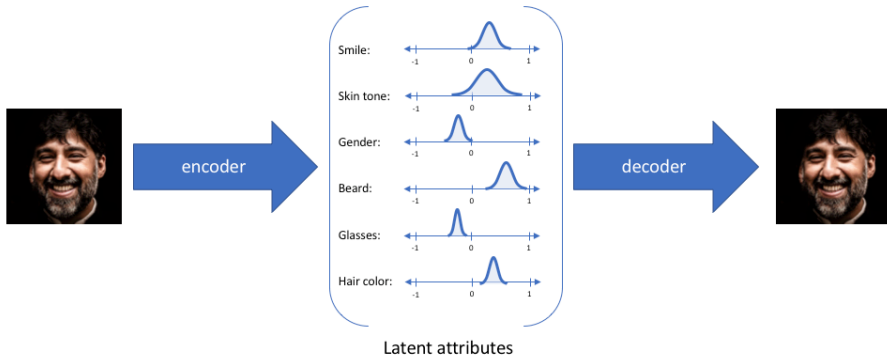


Variational Autoencoder models

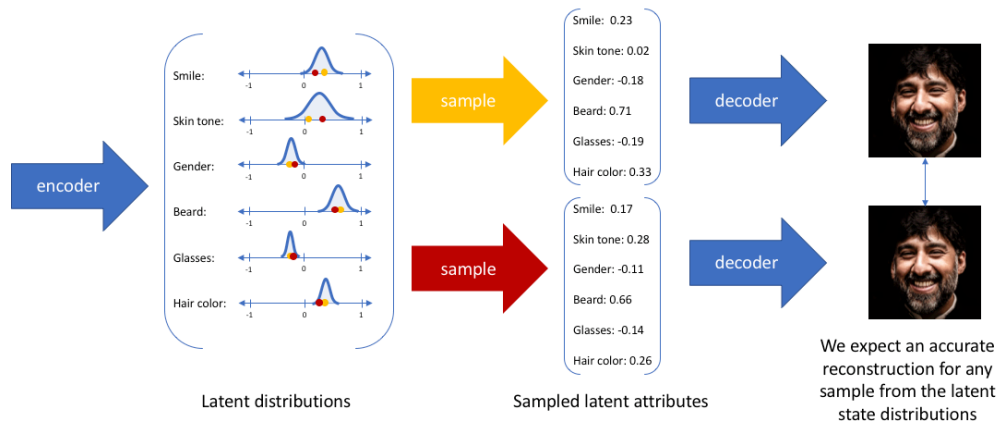
1. An ideal autoencoder will learn descriptive attributes of input to describe an observation in some compressed representation.



1. However, we may prefer to represent each latent attribute as a range of possible values.



1. For any sampling of the latent distributions, we're expecting our decoder model to be able to accurately reconstruct the input.





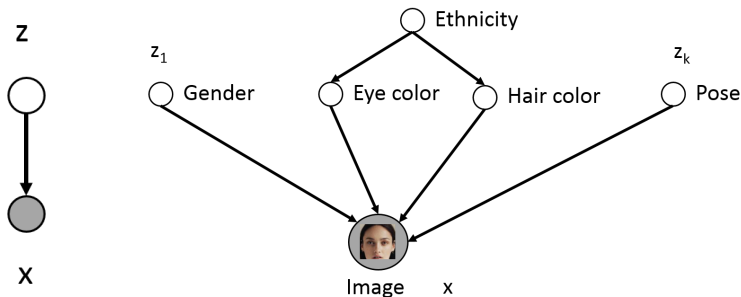
1. Lots of **variability in images x** due to gender, eye color, hair color, pose, etc.
2. However, unless images are annotated, these factors of variation **are not explicitly available (latent)**.



3. **Idea:** explicitly model these factors using **latent variables z** .

⁶Some slides of this lecture are from S. Ermon and A. Grover slides.

1. Consider an image x , and some of its latent factors such as gender, eye color, hair color, pose, etc.



2. Only shaded variables x are observed in the data (pixel values).
3. Latent variables z correspond to high level features.
 - ▶ If z chosen properly, $p(x|z)$ could be much simpler than $p(x)$.
 - ▶ If we trained this model, then we could identify features via $p(z|x)$.
4. **Challenge:** Very difficult to specify these conditionals by hand.



1. Consider an observed variable x , and latent variable z .



2. Instead of modelling $p(x)$ directly, we use an unobserved latent variable z and define $p(x|z)$ for the data.
3. We can use prior distribution $p(z)$ over the z and

$$p(x, z) = p(x|z)p(z).$$

4. Generative process for the observed data x .

$$z \sim p(z)$$

$$x \sim p(x|z).$$



1. Given a set of observed random variables $x = \{x_1, x_2, \dots, x_n\}$ and a set of latent random variables $z = \{z_1, z_2, \dots, z_m\}$, we need to compute the posterior $p(z|x)$.
2. Using Bayes' theorem, we have

$$\begin{aligned} p(z|x) &= \frac{p(x, z)}{p(x)} \\ &= \frac{p(x|z)p(z)}{p(x)} \end{aligned}$$

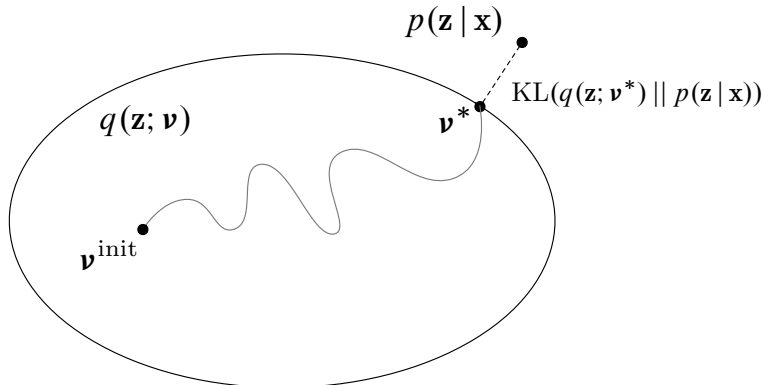
3. $p(x)$ is the marginal density which is also called **evidence**.

$$p(x) = \int_z p(x, z) dz$$

4. For most of the models, computing $p(x)$ is **intractable**. Hence computing $p(z|x)$ is also **intractable**.



1. Since directly computing $p(z|x)$ is intractable, we have to do some **approximate inference**.
2. Variational inference considers a family of parametric distributions that approximates $p(z|x)$.



$\nu = \theta$ is parameter of q .



1. Variational inference leverages optimization to find the best distribution $q(z; \theta)$.
2. In variational inference, we specify a family of distributions \mathcal{Q} over the latent random variables.
3. Each $q(z) \in \mathcal{Q}$ is a candidate approximation to the posterior.
4. Our goal is to find the best candidate that has the smallest KL divergence to the posterior we want to compute.
5. Mathematically, the optimization goal is

$$q^*(z) = \operatorname{argmin}_{q(z) \in \mathcal{Q}} KL(q(z) || p(z|x))$$

where $q^*(\cdot)$ is the best approximation to the posterior in distribution family \mathcal{Q} .



1. To measure the difference between two probability distributions over the same variable x , **Kullback-Leibler divergence** is used.
2. The KL divergence between two distributions p and q with discrete support is defined as

$$KL(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}.$$

3. The KL divergence has the following properties
 - ▶ $KL(p||q) \geq 0$ for all p, q .
 - ▶ $KL(p||q) = 0$ if and only if $q = p$
4. KL divergence is not symmetric, i.e.

$$KL(q||p) \neq KL(p||q)$$



1. The KL divergence between two distributions p and q with discrete support is defined as

$$KL(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)} = \mathbb{E}_p \log \frac{p(x)}{q(x)}.$$

2. It is hard to compute $KL(p||q)$, because taking expectation wrt p is assumed to be intractable.
3. An alternative is the [reverse KL divergence](#), which is

$$KL(q||p) = \sum_x q(x) \log \frac{q(x)}{p(x)} = \mathbb{E}_q \log \frac{q(x)}{p(x)}.$$

4. The main advantage is that computing expectation wrt q is [tractable](#), by choosing a suitable form of q .
5. The above equation is still not tractable because $p(x) = p(x|S)$ is intractable, where S is the given dataset.



1. We'll assume that p is a general undirected model of the following form

$$p(x_1, \dots, x_n; \theta) = \frac{\bar{p}(x_1, \dots, x_n; \theta)}{Z(\theta)} = \frac{1}{Z(\theta)} \prod_k \phi_k(x_k; \theta),$$

where the ϕ_k are the factors and $Z(\theta)$ is the normalization constant.

2. Given this formulation, optimizing $KL(q||p)$ directly is not possible because of the potentially intractable normalization constant $Z(\theta)$.
3. Evaluating $KL(q||p)$ is not possible, because we need to evaluate p .
4. Instead, we work with the following objective (the same form as the KL divergence), but only involves the unnormalized probability $\bar{p}(x) = \prod_k \phi_k(x_k; \theta)$.



1. We use the following objective function

$$J(q) = \sum_x q(x) \log \frac{q(x)}{\bar{p}(x)}.$$

2. This function is not only tractable, it also has the following important property

$$\begin{aligned} J(q) &= \sum_x q(x) \log \frac{q(x)}{\bar{p}(x)} \\ &= \sum_x q(x) \log \frac{q(x)}{p(x)} - \log Z(\theta) \\ &= KL(q\|p) - \log Z(\theta) \end{aligned}$$

3. Since $KL(q\|p) \geq 0$, we get by rearranging terms that

$$\log Z(\theta) = KL(q\|p) - J(q) \geq -J(q).$$



1. Thus, $-J(q)$ is a **lower bound** on the $\log Z(\theta)$.
2. Because of this property, $-J(q)$ is called **variational lower bound** or **evidence lower bound (ELBO)**.
3. ELBO is often written in the form

$$\log Z(\theta) \geq \mathbb{E}_{q(x)}[\log \bar{p}(x) - \log q(x)].$$

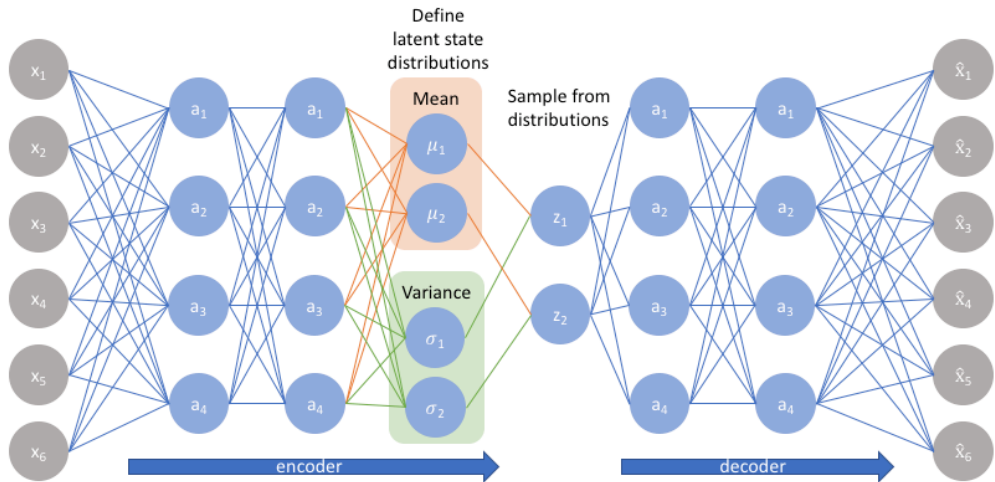
4. The difference between $\log Z(\theta)$ and $-J(q)$ is $KL(q||p)$.
5. Thus, by maximizing **ELBO**, we are minimizing $KL(q||p)$.



1. The idea of VAE is actually less similar to all the autoencoder models, but deeply rooted in graphical models (Kingma and Welling 2014).
2. Instead of mapping the input into a fixed vector, we want to map it into a distribution (in practice, a Gaussian distribution) over encodings.
3. The decoder will then sample an encoding from that probability distribution, and try to reconstruct the original input.
4. This forces the decoder to produce reasonable outputs over a range of different encodings.
5. Since a Gaussian distribution can be parametrized by its mean vector and covariance matrix, we have the encoder output a mean vector μ and a covariance matrix Σ (restricted to a diagonal matrix for simplicity).

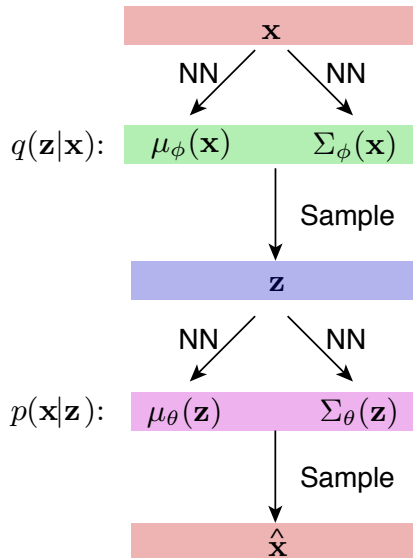


1. The VAE has the following architecture.





1. We can generate through VAE as





1. The VAE introduces a loss other than the reconstruction loss: the KL divergence between the distribution produced by the encoder and a unit Gaussian distribution.
2. We maximize the ELBO.
3. Optimize both networks jointly with SGD.



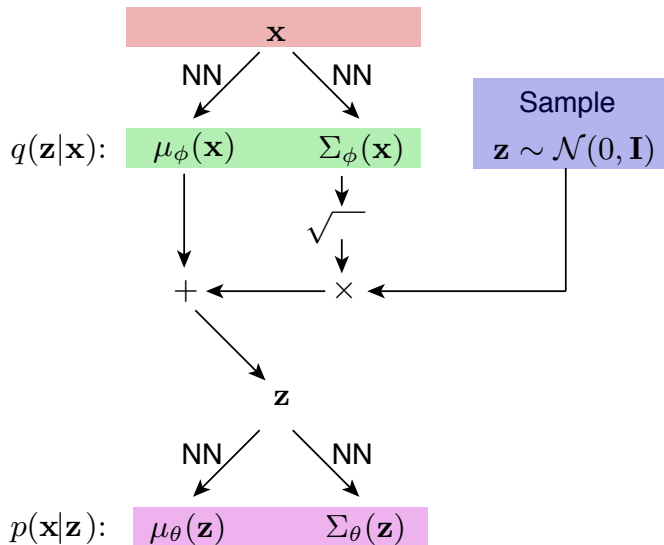
1. Autoencoders are simple to train since you simply have to backpropagate the reconstruction loss across the weights of the network.
2. VAEs are not as simple to optimize though.
3. The key problem is that the sampling operation is not differentiable.
4. This means we cannot propagate the gradients from the reconstruction error to the encoder.
5. Normally we would have to resort to more complicated optimization techniques like REINFORCE.



1. We are able to resolve this problem through the reparameterization trick.
2. The idea behind this trick is to isolate the sampling from the parameter estimation (mean and variance).
3. First, we sample ϵ from a unit Gaussian distribution.
4. We can make the sample to adhere to a Gaussian distribution with mean μ and covariance matrix Σ by transforming it.

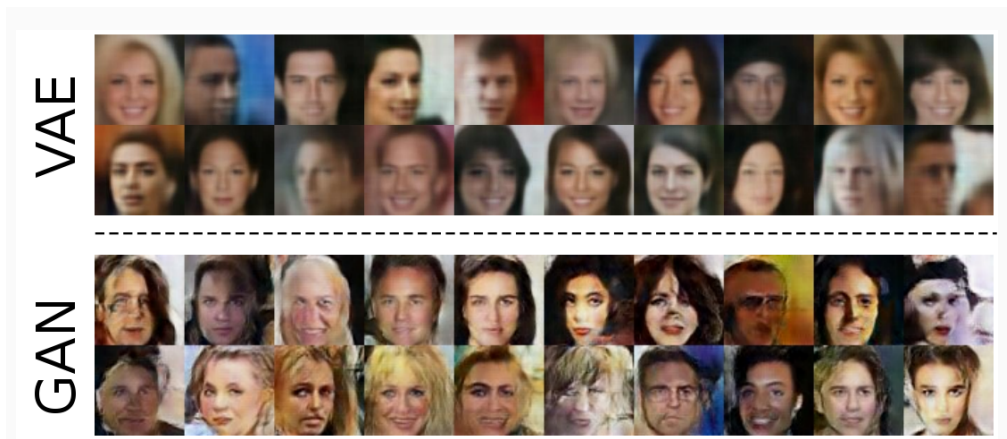


1. We can generate through Reparametrized VAE as





1. The comparison between VAE and GAN.



Normalizing Flow Models



- Given a function of mapping a n -dimensional input vector \mathbf{x} to a m -dimensional output vector, $\mathbf{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$, the Jacobian matrix, \mathbf{J} , is

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- The determinant of a $n \times n$ matrix M is

$$\det(M) = \det \left(\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \right) = \sum_{j_1 j_2 \dots j_n} (-1)^{\tau(j_1 j_2 \dots j_n)} a_{1j_1} a_{2j_2} \cdots a_{nj_n}$$

$\tau(\cdot)$ indicates the signature of a permutation.⁷

⁷Most slides of this section are adopted from <https://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html>

[//lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html](https://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html)



1. Given a random variable z and its known probability density function $z \sim \pi(z)$, we would like to construct a new random variable using a **one-one mapping** function $x = f(z)$.
2. The function f is invertible, so $z = f^{-1}(x)$.
3. The question is how to infer the unknown probability density function of the new variable, $p(x)$?

$$\int_x p(x) dx = \int_z \pi(z) dz = 1 \quad \text{Definition of probability distribution.}$$

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

4. By definition, the integral $\int_z \pi(z) dz$ is the sum of an infinite number of rectangles of infinitesimal width Δz .
5. The height of such a rectangle at position z is the value of the density function $\pi(z)$.



1. When we substitute the variable, $z = f^{-1}(x)$ yields $\frac{\Delta z}{\Delta x} = (f^{-1}(x))'$ and $\Delta z = (f^{-1}(x))' \Delta x$.
2. Here $|(f^{-1}(x))'|$ indicates the ratio between the area of rectangles defined in two different coordinate of variables z and x , respectively.
3. The multivariable version has a similar format:

$$\begin{aligned} \mathbf{z} &\sim \pi(\mathbf{z}), \mathbf{x} = f(\mathbf{z}), \mathbf{z} = f^{-1}(\mathbf{x}) \\ p(\mathbf{x}) &= \pi(\mathbf{z}) \left| \det \left(\frac{d\mathbf{z}}{d\mathbf{x}} \right) \right| \\ &= \pi(f^{-1}(\mathbf{x})) \left| \det \left(\frac{df^{-1}}{d\mathbf{x}} \right) \right| \end{aligned}$$

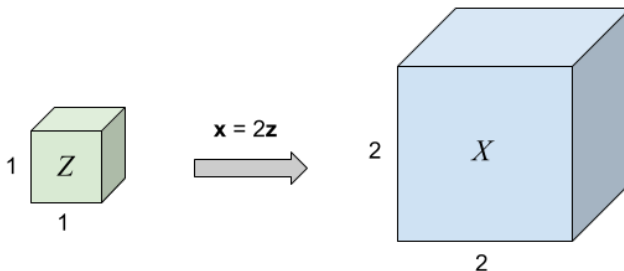
where $\det \left(\frac{\partial f}{\partial \mathbf{z}} \right)$ is the Jacobian determinant of the function f .



1. Consider a random variable Z that is uniformly distributed over the unit cube $\mathbf{z} \in [0, 1]^3$.
2. We can scale Z by a factor of 2 to get a new random variable X ,

$$\mathbf{x} = f(\mathbf{z}) = \mathbf{A}\mathbf{z} = \begin{vmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{vmatrix} \mathbf{z}$$

where X is uniform over a cube with side length 2.





1. How is the density $p(\mathbf{x})$ related to $\pi(\mathbf{z})$?
2. Since every distribution sums to 1 and the unit cube has volume $V_Z = 1$.

$$\pi(\mathbf{z})V_Z = 1$$

3. and $\pi(\mathbf{z}) = 1$ for all \mathbf{z} in the unit cube.
4. The volume of the larger cube is easy to compute: $V_X = 2^3 = 8$.
5. The total probability mass must be conserved, so we can solve for the density of X .

$$p(\mathbf{x}) = \frac{\pi(\mathbf{z})V_Z}{V_X} = \frac{1}{8}.$$

6. The new density is equal to the original density multiplied by the ratio of the volumes.



1. The change of variables formula allows us to tractably compute normalized probability densities when we apply an invertible transformation f .

$$p(\mathbf{x}) = \pi(\mathbf{z}) \left| \det \left(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right) \right| = \pi(\mathbf{z}) \left| \det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) \right|^{-1}$$

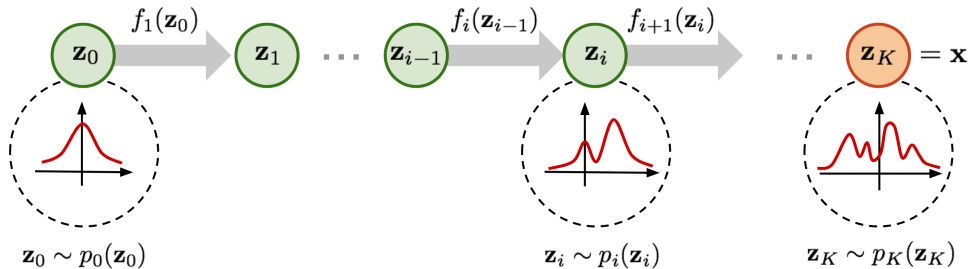
2. The invertible function is just multiplication by a scaling matrix, so the determinant of the Jacobian matrix is easy to compute:

$$\det \left(\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} \right) = \det(\mathbf{A}) = 8.$$



1. Density estimation has several important applications in many machine learning problems.
2. In deep learning models, the embedded probability distribution is expected to be simple enough to calculate the derivative easily and efficiently.
3. This is why Gaussian distribution is often used in latent variable generative models.
4. Normalizing Flow (NF) models are used for better and more powerful distribution approximation (Rezende and Mohamed [2015](#)).
5. A normalizing flow transforms a simple distribution into a complex one by applying a sequence of invertible transformation functions.

1. Normalizing flow transforms a simple distribution into a complex one by applying a sequence of invertible transformation functions (Rezende and Mohamed 2015).





1. From the previous slide, we have

$$\begin{aligned} \mathbf{z}_{i-1} &\sim p_{i-1}(\mathbf{z}_{i-1}) \\ \mathbf{z}_i &= f_i(\mathbf{z}_{i-1}), \text{ thus } \mathbf{z}_{i-1} = f_i^{-1}(\mathbf{z}_i) \\ p_i(\mathbf{z}_i) &= p_{i-1}(f_i^{-1}(\mathbf{z}_i)) \left| \det \left(\frac{df_i^{-1}}{d\mathbf{z}_i} \right) \right| \end{aligned}$$

2. Repeating above, we can do inference using base distribution.

$$\begin{aligned} p_i(\mathbf{z}_i) &= p_{i-1}(\mathbf{z}_{i-1})(f_i^{-1}(\mathbf{z}_i)) \\ &= p_{i-1}(\mathbf{z}_{i-1}) \left| \det \left(\left(\frac{df_i}{d\mathbf{z}_{i-1}} \right)^{-1} \right) \right| \text{ According to the inverse func theorem.} \\ &= p_{i-1}(\mathbf{z}_{i-1}) \left| \det \left(\frac{df_i}{d\mathbf{z}_{i-1}} \right) \right|^{-1} \text{ Using property of Jacobians of invertible func.} \\ \log p_i(\mathbf{z}_i) &= \log p_{i-1}(\mathbf{z}_{i-1}) - \log \left| \det \left(\frac{df_i}{d\mathbf{z}_{i-1}} \right) \right| \end{aligned}$$



1. Given chain of pdfs, we can expand the equation of the output \mathbf{x} step by step until tracing back to the initial distribution \mathbf{z}_0 .

$$\begin{aligned}\mathbf{x} &= \mathbf{z}_K = f_K \circ f_{K-1} \circ \cdots \circ f_1(\mathbf{z}_0) \\ \log p(\mathbf{x}) &= \log \pi_K(\mathbf{z}_K) = \log \pi_{K-1}(\mathbf{z}_{K-1}) - \log \left| \det \left(\frac{df_K}{d\mathbf{z}_{K-1}} \right) \right| \\ &= \log \pi_{K-2}(\mathbf{z}_{K-2}) - \log \left| \det \left(\frac{df_{K-1}}{d\mathbf{z}_{K-2}} \right) \right| \\ &\quad - \log \left| \det \left(\frac{df_K}{d\mathbf{z}_{K-1}} \right) \right| \\ &= \dots \\ &= \log \pi_0(\mathbf{z}_0) - \sum_{i=1}^K \log \left| \det \left(\frac{df_i}{d\mathbf{z}_{i-1}} \right) \right|\end{aligned}$$



1. The path traversed by the random variables $\mathbf{z}_i = f_i(\mathbf{z}_{i-1})$ is the **flow**.
2. The full chain formed by the successive distributions π_i is called a **normalizing flow**.
3. For computation of equation, a transformation function f_i should satisfy two properties:
 - ▶ It is easily invertible.
 - ▶ Its Jacobian determinant is easy to compute.



1. With **normalizing flows**, the exact log-likelihood of input data $\log p(\mathbf{x})$ becomes tractable.
2. The training criterion of flow-based generative model is simply the **negative log-likelihood (NLL)** over the training dataset S .

$$\mathcal{L}(S) = -\frac{1}{|S|} \sum_{\mathbf{x} \in S} \log p(\mathbf{x})$$



1. The RealNVP model implements a normalizing flow by stacking a sequence of invertible bijective transformation functions (Dinh, Sohl-Dickstein, and S. Bengio 2017).
2. In each bijection $f : \mathbf{x} \mapsto \mathbf{y}$, the input dimensions are split into two parts:
 - ▶ The first d dimensions stay same (\mathbf{x}_1);
 - ▶ The second part, $d + 1$ to D dimensions (\mathbf{x}_2) transformed using

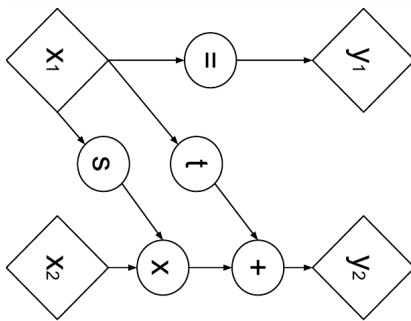
$$\begin{aligned}\mathbf{y}_{1:d} &= \mathbf{x}_{1:d} \\ \mathbf{y}_{d+1:D} &= \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d})\end{aligned}$$

where $s(\cdot)$ and $t(\cdot)$ are scale and translation functions and both map $\mathbb{R}^d \mapsto \mathbb{R}^{D-d}$.
The \odot operation is the element-wise product.



1. This network has

- ▶ Stack many invertible **coupling layers**.
- ▶ Each has simple inverse and determinant





1. This transformation satisfy two properties of flow transformations.

- ▶ It is easily invertible.

$$\begin{cases} \mathbf{y}_{1:d} &= \mathbf{x}_{1:d} \\ \mathbf{y}_{d+1:D} &= \mathbf{x}_{d+1:D} \odot \exp(s(\mathbf{x}_{1:d})) + t(\mathbf{x}_{1:d}) \end{cases}$$

$$\Leftrightarrow \begin{cases} \mathbf{x}_{1:d} &= \mathbf{y}_{1:d} \\ \mathbf{x}_{d+1:D} &= (\mathbf{y}_{d+1:D} - t(\mathbf{y}_{1:d})) \odot \exp(-s(\mathbf{y}_{1:d})) \end{cases}$$

- ▶ Its Jacobian determinant is easy to compute. The Jacobian is a lower triangular matrix.

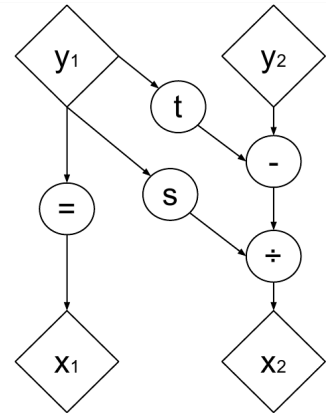
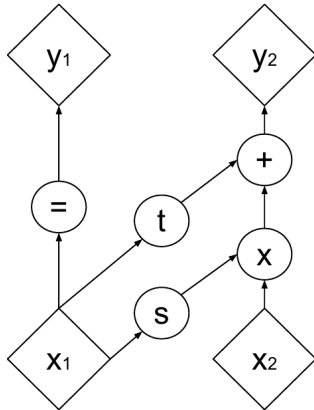
$$\mathbf{J} = \begin{bmatrix} \mathbb{I}_d & \mathbf{0}_{d \times (D-d)} \\ \frac{\partial \mathbf{y}_{d+1:D}}{\partial \mathbf{x}_{1:d}} & \text{diag}(\exp(s(\mathbf{x}_{1:d}))) \end{bmatrix}$$

Hence, the determinant is simply the product of terms on the diagonal.

$$\det(\mathbf{J}) = \prod_{j=1}^{D-d} \exp(s(\mathbf{x}_{1:d})_j) = \exp\left(\sum_{j=1}^{D-d} s(\mathbf{x}_{1:d})_j\right)$$



1. The inverse transformation

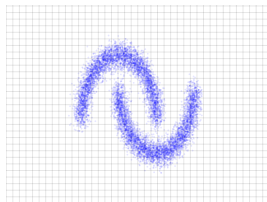




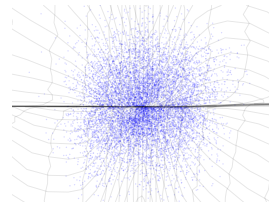
Inference

$$x \sim \hat{p}_X$$
$$z = f(x)$$

Data space \mathcal{X}

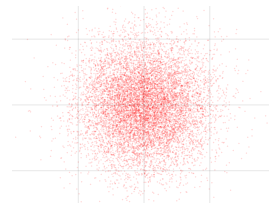
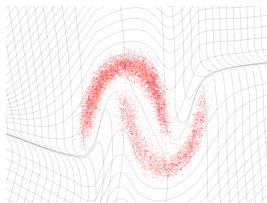


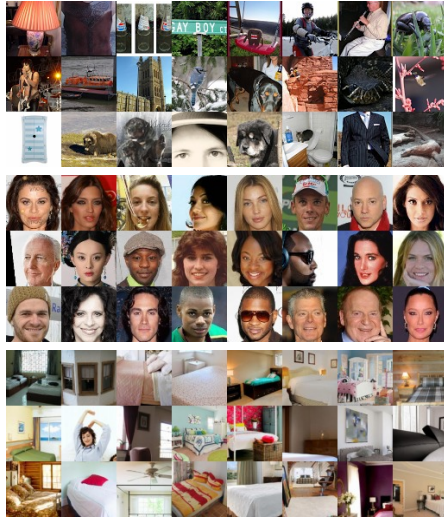
Latent space \mathcal{Z}



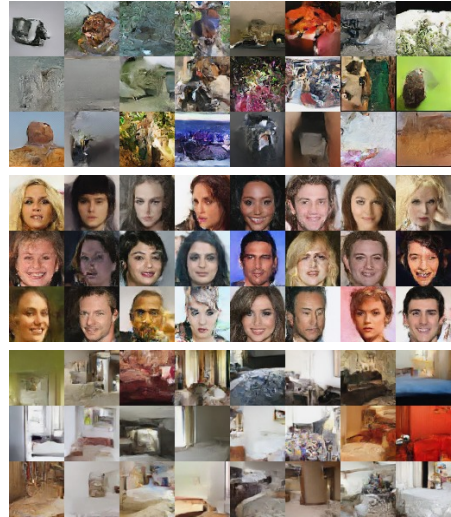
Generation

$$z \sim p_Z$$
$$x = f^{-1}(z)$$





Dataset



samples from model



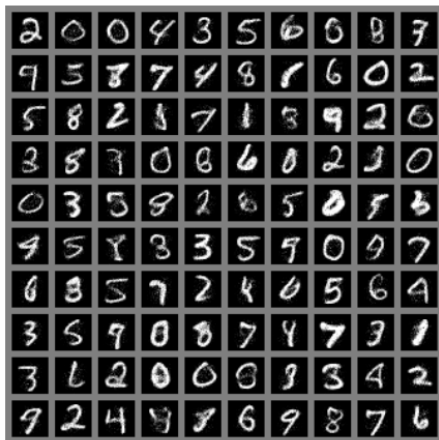
1. The NICE model is a predecessor of Real NVP (Dinh, Krueger, and Y. Bengio 2015).
2. The transformation in NICE is the affine coupling layer without the scale term, known as additive coupling layer.

$$\begin{cases} \mathbf{y}_{1:d} &= \mathbf{x}_{1:d} \\ \mathbf{y}_{d+1:D} &= \mathbf{x}_{d+1:D} + m(\mathbf{x}_{1:d}) \end{cases} \Leftrightarrow \begin{cases} \mathbf{x}_{1:d} &= \mathbf{y}_{1:d} \\ \mathbf{x}_{d+1:D} &= \mathbf{y}_{d+1:D} - m(\mathbf{y}_{1:d}) \end{cases}$$

3. m is an arbitrarily complex function, in this case a ReLU MLP.
4. Additive layers have unit Jacobian determinant, and their composition will necessarily have unit Jacobian determinant too.
5. NICE includes a diagonal scaling matrix \mathbf{S} as the top layer.
6. Final layer of NICE applies a rescaling transformation $x_i = s_j z_j$ and inverse mapping $z_j = \frac{x_j}{s_j}$.
7. Jacobian of forward mapping:

$$J = \text{diag}(\mathbf{S})$$

$$\det(J) = \prod_i s_j.$$



(a) Model trained on MNIST



(b) Model trained on TFD

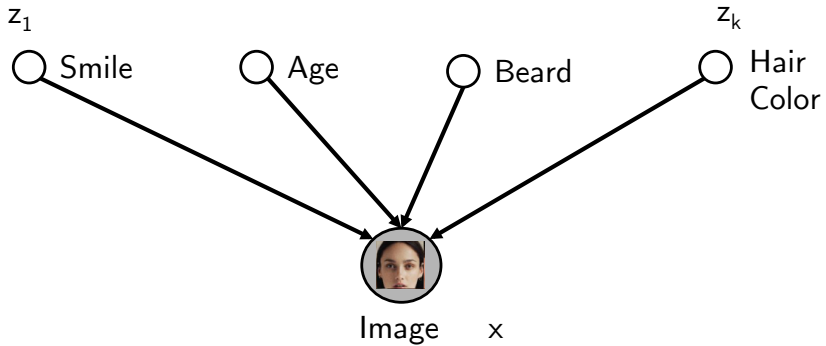


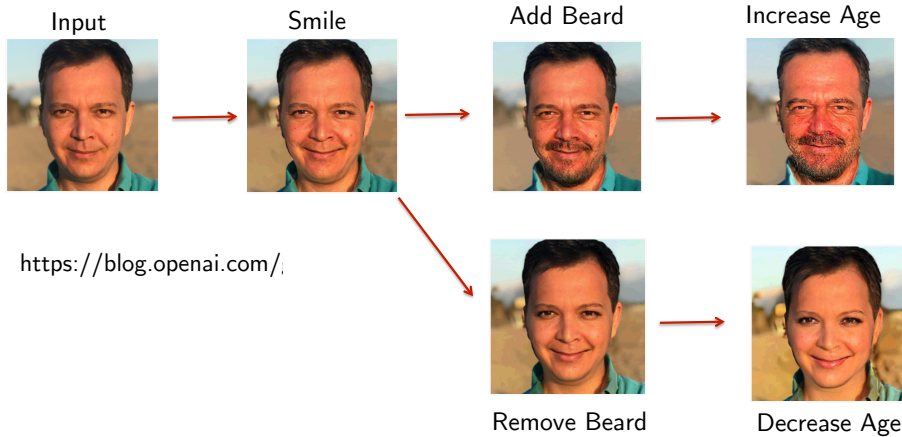
1. The Glow model extends NICE and RealNVP, and simplifies the architecture by replacing the reverse permutation operation on the channel ordering with invertible 1×1 convolutions (Kingma and Dhariwal 2018).
2. There are three substeps in one step of flow in Glow.
 - ▶ Activation normalization (short for **actnorm**):

Activation normalization

- ▶ Actnorm layer performs an affine transformation of the activations using a scale and bias parameter per channel.
 - ▶ These parameters are initialized such that the post-actnorm activations per-channel have zero mean and unit variance.
 - ▶ After initialization, the scale and bias are treated as regular trainable parameters that are independent of the data.
-
- ▶ Invertible 1×1 convolution: Instead of fixed ordering, 1×1 convolution is used.
 - ▶ Affine coupling layer (Same as in RealNVP)

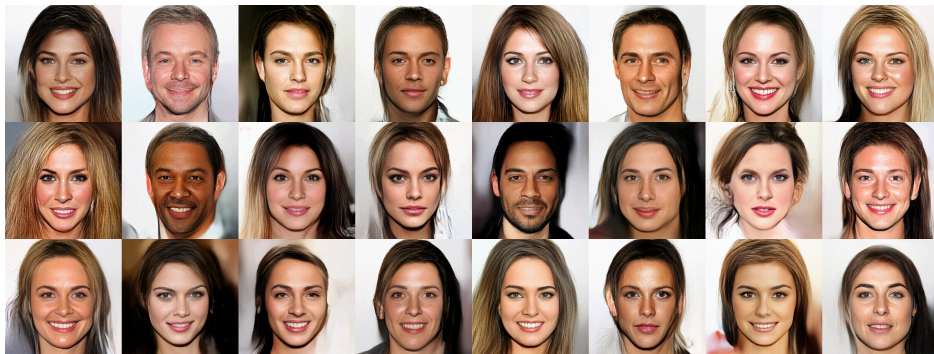
1. Latent factors







Synthetic celebrities sampled from Glow model



Random samples from the Glow model.

See also <https://openai.com/blog/glow/>.



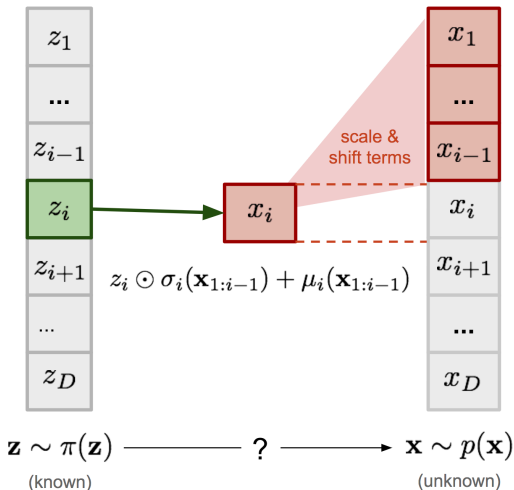
1. In **autoregressive models**, the probability of observing x_i is conditioned on x_1, \dots, x_{i-1} and the product of these conditional probabilities gives us the probability of observing the full sequence:

$$p(\mathbf{x}) = \prod_{i=1}^D p(x_i | x_1, \dots, x_{i-1}) = \prod_{i=1}^D p(x_i | \mathbf{x}_{1:i-1})$$

2. If a flow transformation in a normalizing flow is framed as an autoregressive model, the model is an **autoregressive flow**.



1. **MAF** is a type of normalizing flows, where the transformation layer is built as an autoregressive neural network (Papamakarios, Murray, and Pavlakou 2017).





1. Given two random variables $\mathbf{z} \sim \pi(\mathbf{z})$ and $\mathbf{x} \sim p(\mathbf{x})$, and the probability density function $\pi(\mathbf{z})$ is known, MAF aims to learn $p(\mathbf{x})$.
2. MAF generates each x_i , conditioned on the past dimensions $\mathbf{x}_{1:i-1}$.
 - ▶ Data generation, producing a new \mathbf{x} .

$$x_i = z_i \exp \alpha_i + \mu_i$$

where

$$p(x_i | \mathbf{x}_{1:i-1}) = \mathcal{N}(x_i | \mu_i, (\exp \alpha_i)^2)$$

$$\mu_i = f_{\mu_i}(\mathbf{x}_{1:i-1})$$

$$\alpha_i = f_{\alpha_i}(\mathbf{x}_{1:i-1})$$

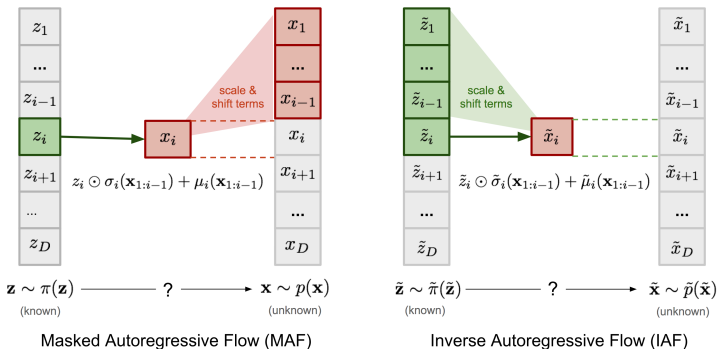
$$z_i \sim \mathcal{N}(0, 1)$$

- ▶ Density estimation, given a known \mathbf{x} .

$$p(\mathbf{x}) = \prod_{i=1}^D p(x_i | \mathbf{x}_{1:i-1})$$



- IAF models the conditional probability of the target variable as an autoregressive model too, but with a reversed flow (for efficient sampling process) (Kingma, Salimans, and Welling 2016).



- In IAF, the nonlinear shift/scale statistics are computed using the previous noise variates $\mathbf{z}_{1:i-1}$, instead of the data samples:

$$x_i = z_i \exp \alpha_i + \mu_i$$

$$\mu_i = f_{\mu_i}(\mathbf{z}_{1:i-1})$$

$$\alpha_i = f_{\alpha_i}(\mathbf{z}_{1:i-1})$$



1. The reverse transformation in MAF is

$$z_i = \frac{x_i - \mu_i(\mathbf{x}_{1:i-1})}{\sigma_i(\mathbf{x}_{1:i-1})} = -\frac{\mu_i(\mathbf{x}_{1:i-1})}{\sigma_i(\mathbf{x}_{1:i-1})} + x_i \odot \frac{1}{\sigma_i(\mathbf{x}_{1:i-1})}$$

2. If we consider

$$\tilde{\mathbf{x}} = \mathbf{z}, \tilde{p}(\cdot) = \pi(\cdot), \tilde{\mathbf{x}} \sim \tilde{p}(\tilde{\mathbf{x}})$$

$$\tilde{\mathbf{z}} = \mathbf{x}, \tilde{\pi}(\cdot) = \rho(\cdot), \tilde{\mathbf{z}} \sim \tilde{\pi}(\tilde{\mathbf{z}})$$

$$\tilde{\mu}_i(\tilde{\mathbf{z}}_{1:i-1}) = \tilde{\mu}_i(\mathbf{x}_{1:i-1}) = -\frac{\mu_i(\mathbf{x}_{1:i-1})}{\sigma_i(\mathbf{x}_{1:i-1})}$$

$$\tilde{\sigma}_i(\tilde{\mathbf{z}}_{1:i-1}) = \tilde{\sigma}_i(\mathbf{x}_{1:i-1}) = \frac{1}{\sigma_i(\mathbf{x}_{1:i-1})}$$

3. Then, $\tilde{x}_i \sim p(\tilde{x}_i | \tilde{\mathbf{z}}_{1:i}) = \tilde{z}_i \odot \tilde{\sigma}_i(\tilde{\mathbf{z}}_{1:i-1}) + \tilde{\mu}_i(\tilde{\mathbf{z}}_{1:i-1})$, where $\tilde{\mathbf{z}} \sim \tilde{\pi}(\tilde{\mathbf{z}})$
4. IAF intends to estimate the probability density function of $\tilde{\mathbf{x}}$ given that $\tilde{\pi}(\tilde{\mathbf{z}})$ is already known.

Evaluating deep generative models



1. Evaluation of generative models is tricky
2. The key questions is about underlying task of the generative model.
 - ▶ Density estimation
 - ▶ Sampling / generation
 - ▶ Latent representation learning
 - ▶ More than one task.
3. How do we evaluate generative models?

Example (Evaluating density estimation)

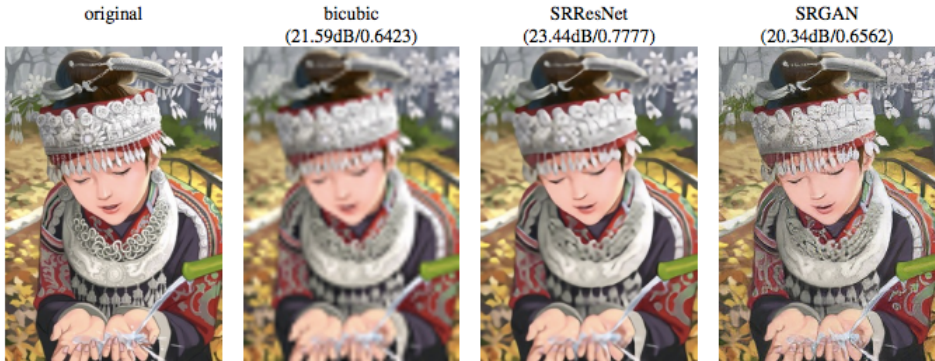
When the given model has tractable likelihood, the evaluation is straightforward.

- ▶ Split dataset into **train**, **validation**, and **test** sets.
- ▶ Evaluate gradients based on the **train set**.
- ▶ Tune hyper-parameters based on the **validation set**.
- ▶ Evaluate generalization by measuring likelihoods on the **test set**.



1. We have a dataset that sampled from p_{data} and generated samples from p_g .
2. Evaluating deep generative models (DGM) is hard because
 - ▶ the distributions of interest are often high dimensional,
 - ▶ the likelihood functions are not always available or easily computable.
3. A common way to evaluate a DGM is to measure how close p_{data} is to p_g .
4. Since **sample complexity** of traditional measure such as **KL divergence** or **Wasserstein distance** is exponential in the dimensionality of the distribution, they cannot be used for real world distributions.
5. The **reduced sample complexity** comes at the cost of **reduced discriminative power**.
6. These metrics cannot tell the difference between a model that memorizes the training data and a model that generalizes.

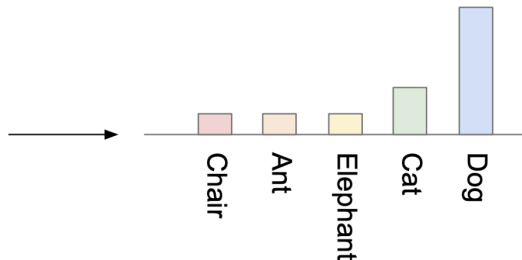
1. Some generative models such as VAE and GAN have **intractable** likelihoods.
2. For example, in VAE we can compare the **evidence lower bounds** (ELBO) to **log-likelihoods**.
3. For general case, **kernel density estimates** only via samples can be used.
4. Consider the following generated images, which of them is better?





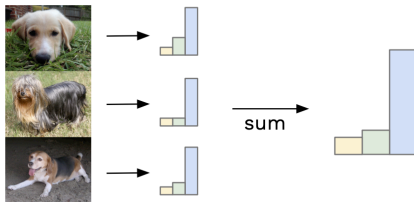
1. One intuitive metric of performance can be obtained by having **human annotators judge** the visual quality of samples.
2. This process can be automated using Amazon Mechanical Turk (Salimans, I. J. Goodfellow, et al. [2016](#)).
3. The task is to ask annotators to distinguish between **generated data** and **real data**.
4. For **MNIST** dataset and GAN model, annotators were able to distinguish samples in **52.4%** of cases (2000 votes total), where **50%** would be obtained by random guessing.
5. For **CIFAR-10** dataset and GAN model, annotators were able to distinguish samples in **78.7%** of cases.
6. A downside of using human annotators is that the metric varies depending on the setup of the task and the motivation of the annotators.
7. Also, results change drastically when we give annotators feedback about their mistakes.
8. By learning from such feedback, annotators are better able to point out the flaws in generated images, giving a more pessimistic quality assessment.

1. The **inception score** takes a list of images and returns a single number, the **score**.
2. The score is a measure of how realistic the output of a generative model (GAN) is.
3. The score measures two things simultaneously:
 - ▶ The images have variety.
 - ▶ Each image distinctly looks like something.
4. If both things are true, **the score will be high**; otherwise, **the score will be low**.
5. The lower bound of this score is zero and the upper bound is ∞ .
6. The inception score takes its name from the **Inception classifier**, an image classification network from Google.
7. Classifier takes an image, and returns probability distribution of labels for image.

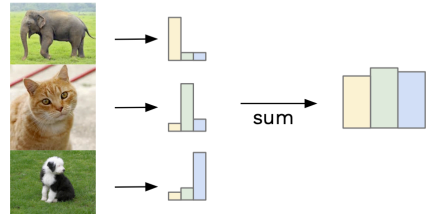


1. If image contains just one well-formed thing, then output of classifier is a narrow distribution.
2. If image is a jumble, or contains multiple things, it's closer to the **uniform** distribution of many similar height bars.
3. The next step is combine the label probability distributions for many of generated images (50,000 images).
4. By summing the label distributions of our images, a new label distribution (**marginal distribution**) will be obtained.
5. The marginal distribution tells the variety in the generator's output:

Similar labels sum to give focussed distribution



Different labels sum to give uniform distribution



6. The final step is to combine these two different things into one single score.



1. The final step is to combine these two different things into one single score.
2. By comparing **label distribution** with **marginal label distribution** for images, a score will be obtained that shows how much those two distributions differ.
3. The more they differ, the higher a score we want to give, and this is the inception score.
4. To produce the inception score, the KL divergence between **label distribution** and **marginal label distribution** is used.
 - ▶ Construct an estimator of the **Inception Score** from samples $\mathbf{x}^{(i)}$ by constructing an empirical marginal class distribution,

$$\hat{p}(y) = \frac{1}{m} \sum_{i=1}^m p(y \mid \mathbf{x}^{(i)})$$

- ▶ Then an approximation to the **expected KLdivergence** is computed by

$$IS(G) \approx \exp \left(\frac{1}{m} \sum_{i=1}^m D_{KL}(p(y \mid \mathbf{x}^{(i)}) \parallel \hat{p}(y)) \right)$$



1. Several metrics have been proposed for evaluation of generative models (Thanh-Tung and Tran 2020).
2. Divergence based evaluation metrics
 - ▶ Inception score
 - ▶ Fréchet inception distance
 - ▶ Neural net divergence
3. Precision-Recall based evaluation metrics
 - ▶ k -means based Precision-Recall
 - ▶ k -NN based Precision-Recall
4. Other evaluation metrics
 - ▶ Metrics for class-conditional models
 - ▶ Topological/Geometrical approaches
 - ▶ Non-parametric approaches

Summary



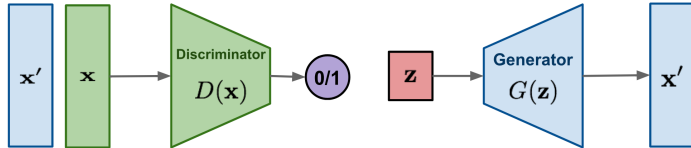
1. Marginal distribution on x obtained by integrating out z

$$p(z) = \mathcal{N}(z; 0, I)$$
$$p_{\theta}(x) = \int_z p(z) p(x | f_{\theta}(z))$$

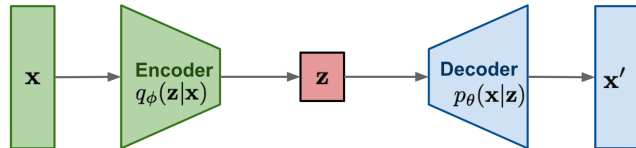
2. **Problem:** Evaluation of $p_{\theta}(x)$ intractable due to integral involving flexible non-linear deep net $f_{\theta}(z)$.
3. **Solutions:** by different unsupervised deep learning paradigms
 - ▶ **Avoid integral:** Generative adversarial networks (GAN)
 - ▶ **Approximate integral:** Variational autoencoders (VAE)
 - ▶ **Tractable integral:** constrain $f_{\theta}(z)$ to invertible **flow**. Please read (Kobyzev, Prince, and Brubaker 2020).
 - ▶ **Avoid latent variables:** autoregressive models



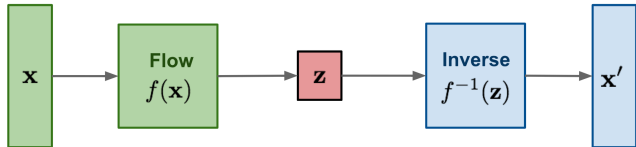
GAN: minimax the classification error loss.



VAE: maximize ELBO.



Flow-based generative models: minimize the negative log-likelihood



Reading







1. Chapter 10 of [Deep Learning Book](#)⁸

⁸Ian Goodfellow, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.



-  Arjovsky, Martin and Leon Bottou (2017). “Towards Principled Methods for Training Generative Adversarial Networks”. In: *International Conference on Learning Representations*.
-  Arjovsky, Martin, Soumith Chintala, and Léon Bottou (2017). “Wasserstein GAN”. In: *ArXiv*.
-  Dinh, Laurent, David Krueger, and Yoshua Bengio (2015). “NICE: Non-linear Independent Components Estimation”. In: *International Conference on Learning Representations*.
-  Dinh, Laurent, Jascha Sohl-Dickstein, and Samy Bengio (2017). “Density estimation using Real NVP”. In: *International Conference on Learning Representations*.
-  Duda, Richard O., Peter E. Hart, and David G. Stork (2001). *Pattern classification, 2nd Edition*. Wiley.
-  Dumoulin, Vincent and Francesco Visin (2016). “A guide to convolution arithmetic for deep learning”. In: *ArXiv*. eprint: [1603.07285](https://arxiv.org/abs/1603.07285).



-  Gan, Zhe et al. (2015). “Learning Deep Sigmoid Belief Networks with Data Augmentation”. In: *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, AISTATS*.
-  Germain, Mathieu et al. (2015). “MADE: Masked Autoencoder for Distribution Estimation”. In: *Proceedings of the 32nd International Conference on Machine Learning*.
-  Goodfellow, Ian J. et al. (2014). “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems*, pp. 2672–2680.
-  Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. MIT Press.
-  Karras, Tero et al. (2018). “Progressive Growing of GANs for Improved Quality, Stability, and Variation”. In: *International Conference on Learning Representations*.
-  Khajenezhad, Ahmad, Hatef Madani, and Hamid Beigy (2021). “Masked Autoencoder for Distribution Estimation on Small Structured Data Sets”. In: *IEEE Transactions on Neural Networks and Learning Systems*.




-  Kingma, Diederik P. and Prafulla Dhariwal (2018). “Glow: Generative Flow with Invertible 1x1 Convolutions”. In: *Advances in Neural Information Processing Systems*, pp. 10236–10245.
-  Kingma, Diederik P., Tim Salimans, and Max Welling (2016). “Improving Variational Inference with Inverse Autoregressive Flow”. In:
-  Kingma, Diederik P. and Max Welling (2014). “Auto-Encoding Variational Bayes”. In: *Proc. of the 2nd Int. Conf. on Learning Representations*.
-  Kobyzev, Ivan, Simon J.D. Prince, and Marcus A. Brubaker (2020). “Normalizing Flows: An Introduction and Review of Current Methods”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*.
-  Larochelle, Hugo and Iain Murray (2011). “The Neural Autoregressive Distribution Estimator”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS*.
-  Mirza, Mehdi and Simon Osindero (2014). “Conditional Generative Adversarial Nets”. In: *arXiv*.




-  Oord, Aäron van den, Sander Dieleman, et al. (2016). “WaveNet: A Generative Model for Raw Audio”. In: *The 9th ISCA Speech Synthesis Workshop*.
-  Oord, Aäron van den, Nal Kalchbrenner, Lasse Espeholt, et al. (2016). “Conditional Image Generation with PixelCNN Decoders”. In: *Advances in Neural Information Processing Systems*.
-  Oord, Aäron van den, Nal Kalchbrenner, and Koray Kavukcuoglu (2016). “Pixel Recurrent Neural Networks”. In: *Proceedings of the 33rd International Conference on Machine Learning*.
-  Papamakarios, George, Iain Murray, and Theo Pavlakou (2017). “Masked Autoregressive Flow for Density Estimation”. In: *Advances in Neural Information Processing Systems*, pp. 2338–2347.
-  Radford, Alec, Luke Metz, and Soumith Chintala (2016). “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *International Conference on Learning Representations*.



-  Rezende, Danilo Jimenez and Shakir Mohamed (2015). “Variational Inference with Normalizing Flows”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37, pp. 1530–1538. URL: <http://proceedings.mlr.press/v37/rezende15.html>.
-  Salakhutdinov, Ruslan and Hugo Larochelle (2010). “Efficient Learning of Deep Boltzmann Machines”. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*. Vol. 9, pp. 693–700.
-  Salimans, Tim, Ian J. Goodfellow, et al. (2016). “Improved Techniques for Training GANs”. In: *Advances in Neural Information Processing Systems*, pp. 2226–2234.
-  Salimans, Tim, Andrej Karpathy, et al. (2017). “PixelCNN++: Improving the PixelCNN with Discretized Logistic Mixture Likelihood and Other Modifications”. In: *International Conference on Learning Representations, ICLR*.
-  Thanh-Tung, Hoang and Truyen Tran (2020). “Toward a Generalization Metric for Deep Generative Models”. In: *arXiv abs/2011.00754*.
-  Uribe, Benigno, Marc-Alexandre Côté, et al. (2016). “Neural Autoregressive Distribution Estimation”. In: *Journal of Machine Learning Research* 17.205, pp. 1–37.



-  Uria, Benigno, Iain Murray, and Hugo Larochelle (2013). “RNADE: The real-valued neural autoregressive density-estimator”. In: *Advances in Neural Information Processing Systems*, pp. 2175–2183.

