Advanced Deep Learning

Deep Generative Models-2

U Kang
Seoul National University

Outline

- ☑ Boltzmann Machines
- ☑ Restricted Boltzmann Machines
- ☑ Deep Belief Networks
- ☑ Deep Boltzmann Machines
- ➡ ☐ **Back-Propagation through Random Operations**
- ☐ Directed Generative Nets

Motivation

- Typical neural networks implement a deterministic transformation of some input variables x
- In developing generative models, we often wish to extend neural networks to implement stochastic transformation of x
- One simple way to do this is to augment the neural network with extra inputs z that are sampled from simple prob. distribution such as uniform or Gaussian
- The function f(x,z) will appear stochastic to an observer who does not have access to z
- Provided that f is continuous and differentiable, we can compute gradients for training using back-propagation

Back-Prop Through Sampling

- Consider the operation consisting of drawing samples $y \sim N(\mu, \sigma^2)$
- It may seem counterintuitive to differentiate y wrt the parameter of its distribution (μ and σ^2)
- However, we can use a reparameterization trick, which is to rewrite the sampling process as transforming an underlying random value $z \sim N(0,1)$ to obtain a sample from the desired distribution: $y = \mu + \sigma z$
- Now we can back-propagate through the sampling operation, by regarding it as a deterministic operation with an extra input z
- The extra input should be a random variable which is not a function of any of the parameter we differentiate against y

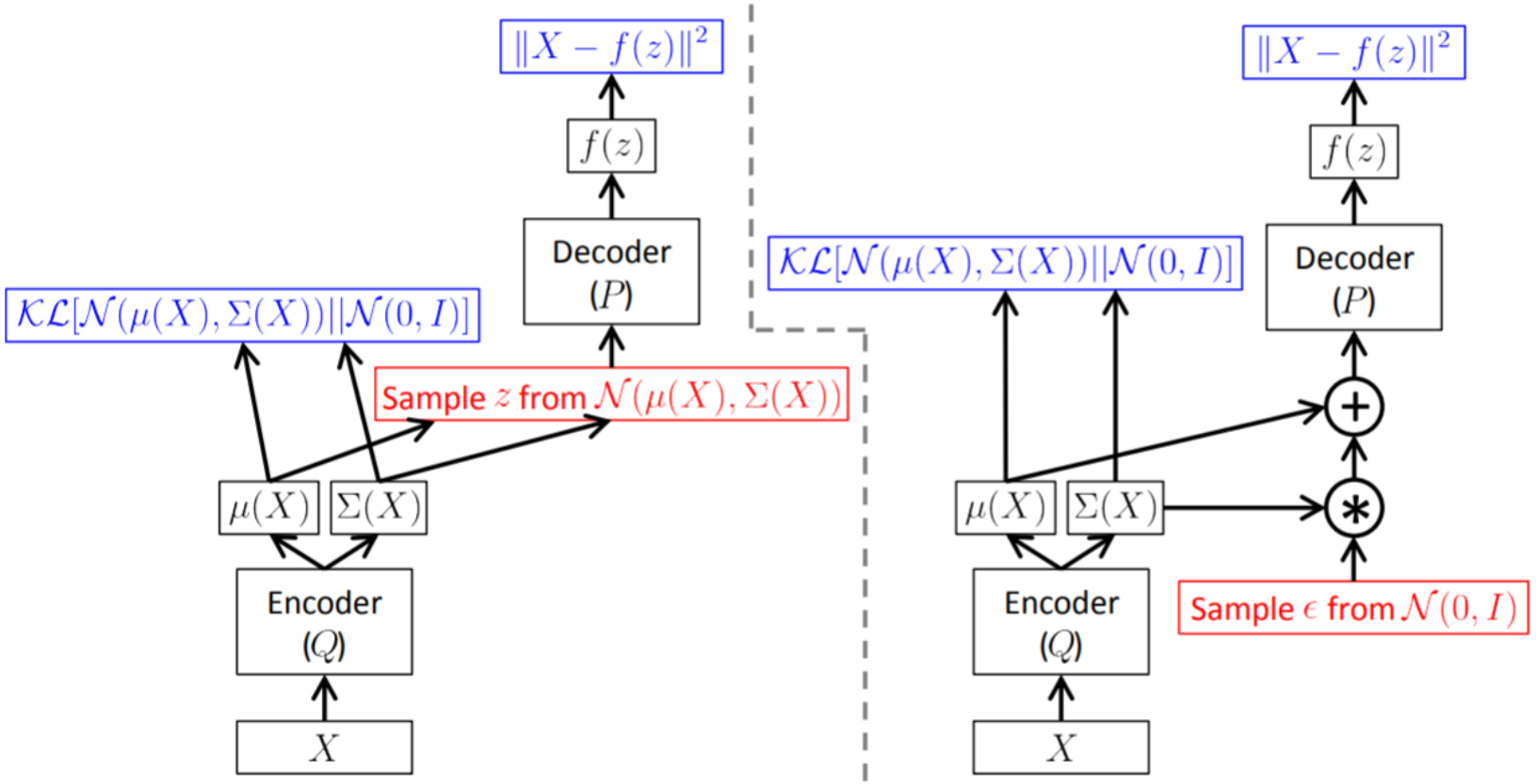
Back-Prop Through Sampling

- The result (e.g., $\frac{\partial y}{\partial \mu}$ or $\frac{\partial y}{\partial \sigma}$) tells us how an infinitesimal change in μ or σ would change the output if we could repeat the sampling operation again with the same value of z
- Being able to back-propagating through sampling operation allows us to incorporate it into a larger graph
- We can build elements of the graph on top of the output of the sampling distribution; we also can build elements of the graph whose outputs are the inputs or the parameters of the sampling operation
- E.g., we could build a large graph with $\mu = f(x; \theta)$ and $\sigma = g(x; \theta)$. Then, we can use back-prop to compute $\nabla_\theta J(y)$

Reparametrization

- Reparametrization = stochastic back-propagation = perturbation analysis
  - Consider a probability distribution $p(y|\theta, x) = p(y|w)$ where w is a variable containing both parameters θ and the input x
  - We rewrite y$\sim p(y|w)$ as $y = f(z; w)$ where z is a source of randomness
  - We may the differentiate y with respect to w using back-propagation applied to f, so long as f is continuous and differentiable
  - Crucially, w must not be a function of z, and z must not be a function of w

Example: VAE



[Doersch, "Tutorial on Variational Autoencoders", 2016]

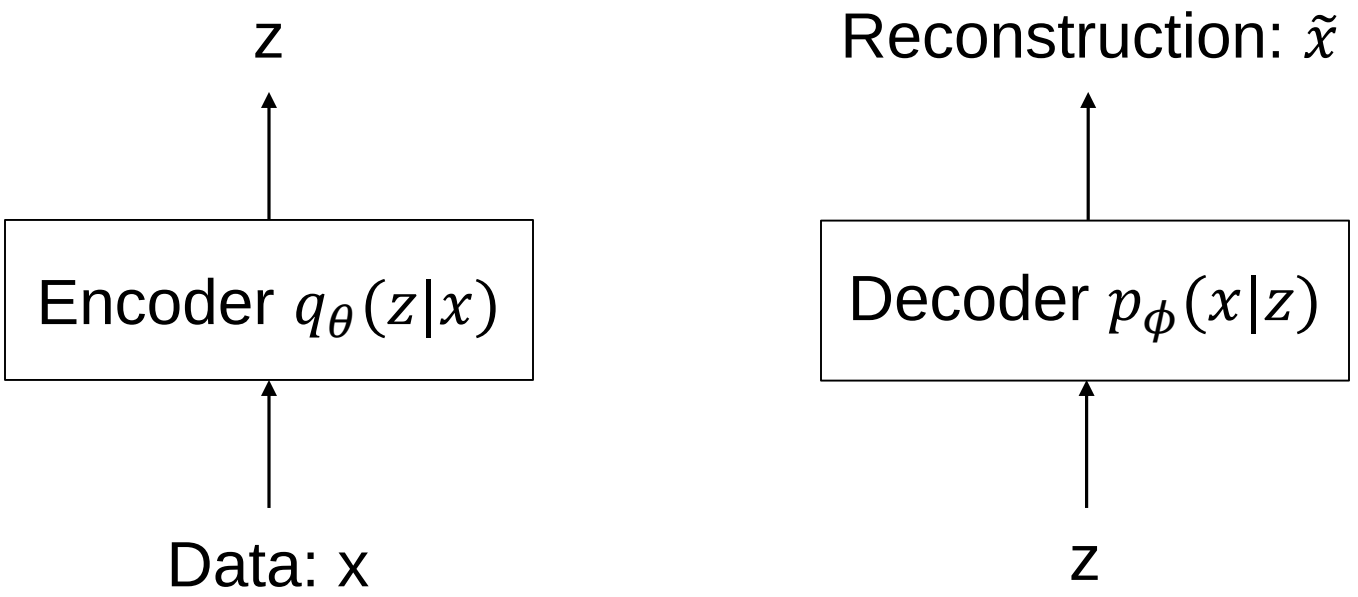
Outline

- ☑ Boltzmann Machines
- ☑ Restricted Boltzmann Machines
- ☑ Deep Belief Networks
- ☑ Deep Boltzmann Machines
- ☑ Other Boltzmann Machines
- ☑ Back-Propagation through Random Operations
- ➡ ☐ **Directed Generative Nets**
  - ➡ VAE
  - GAN

Variational Autoencoder (VAE)

- A generative model that uses learned approximate inference and can be trained purely with gradient-based method
- To generate a sample from the model, VAE first draws a sample z from the code distribution $p_{model}(z)$; the sample then is run through a differentiable generator network g(z)
- Finally, x is sampled from a distribution $p_{model}\big(x; g(z)\big) = p_{model}(x|z)$
- However, during training, the approximate inference network (or encoder) $q(z|x)$ is used to obtain z and $p_{model}(x|z)$ is viewed as a decoder network

Variational Autoencoder (VAE)



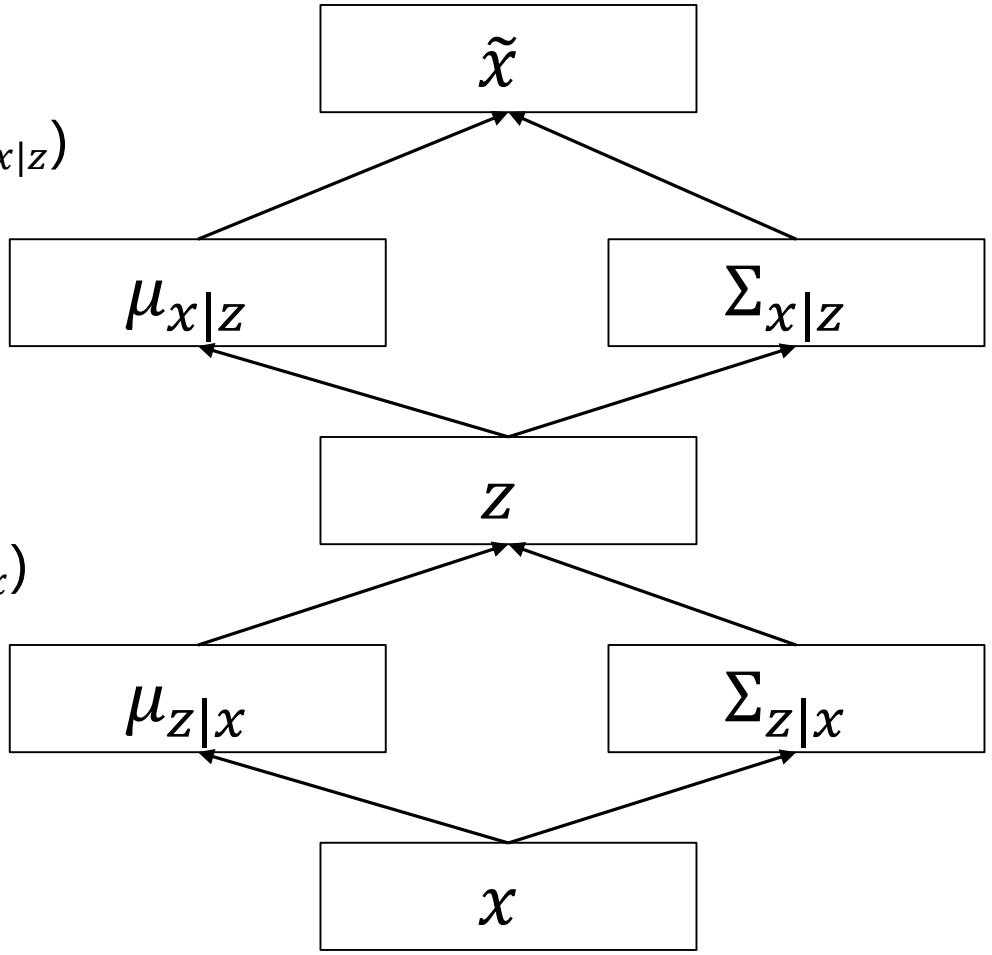
Variational Autoencoder (VAE)

Sample x|z from $x|z \sim N(\mu_{x|z}, \Sigma_{x|z})$

Decoder network
$p_\theta(x|z)$

Sample z from $z|x \sim N(\mu_{z|x}, \Sigma_{z|x})$

Encoder network
$q_\phi(z|x)$



Objective Function of VAE

- VAE is trained by maximizing ELBO $L(q)$ associated with any data point x

$$\mathcal{L}(q) = \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x}) + \mathcal{H}(q(\mathbf{z} \mid \boldsymbol{x})) \quad (1)$$

$$= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z}) - D_{\text{KL}}(q(\mathbf{z} \mid \boldsymbol{x}) || p_{\text{model}}(\mathbf{z})) \quad (2)$$

$$\leq \log p_{\text{model}}(\boldsymbol{x}).$$

  - In eq. (1), the first term is the joint log-likelihood of the visible and the hidden variables. The second term is the entropy of the approximate posterior. When q is chosen to be a Gaussian distribution, maximizing L encourages to place high probability mass on many z values that could have generated x, rather than collapsing to a single point estimate of the most likely value

Objective Function of VAE

- VAE is trained by maximizing ELBO $L(q)$ associated with any data point x

$$\mathcal{L}(q) = \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x}) + \mathcal{H}(q(\mathbf{z} \mid \boldsymbol{x})) \quad (1)$$

$$= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z}) - D_{\text{KL}}(q(\mathbf{z} \mid \boldsymbol{x}) || p_{\text{model}}(\mathbf{z})) \quad (2)$$

$$\leq \log p_{\text{model}}(\boldsymbol{x}).$$

  - In eq. (2), the first term can be viewed as the reconstruction log-likelihood found in other autoencoders. The second term tries to make the approximate posterior distribution q(z|x) and the model prior $p_{model}(z)$ to be similar

Main Idea of VAE

- Train a parametric encoder that produces the parameters of q
- So long as z is a continuous variable, we then back-propagate through samples of z drawn from from $q(z|x) = q(z; f(x; \theta))$ in order to update θ
  - Back-prop through random operation is used in the middle
- Learning then consists sole of maximizing L wrt the parameters of the encoder and decoder. All of the expectations in L can be approximated by Monte Carlo sampling

Main Idea of VAE

- The encoder and decoder are feedforward neural networks, where the outputs are parameters of Gaussian
  - I.e., $\mathrm{P}(z|x) \sim N(\mu_{z|x}, \Sigma_{z|x})$ where $\mu_{z|x}$ and $\Sigma_{z|x}$ are outputs of the encoder network; $\mathrm{P}(x|z) \sim N(\mu_{x|z}, \Sigma_{x|z})$ where $\mu_{x|z}$ and $\Sigma_{x|z}$ are outputs of the decoder network
- Also, z is modeled as a simple Gaussian $N(0, I)$. This makes generating samples easily
  - $q(z|x)$ is modeled as similar as possible to $N(0, I)$

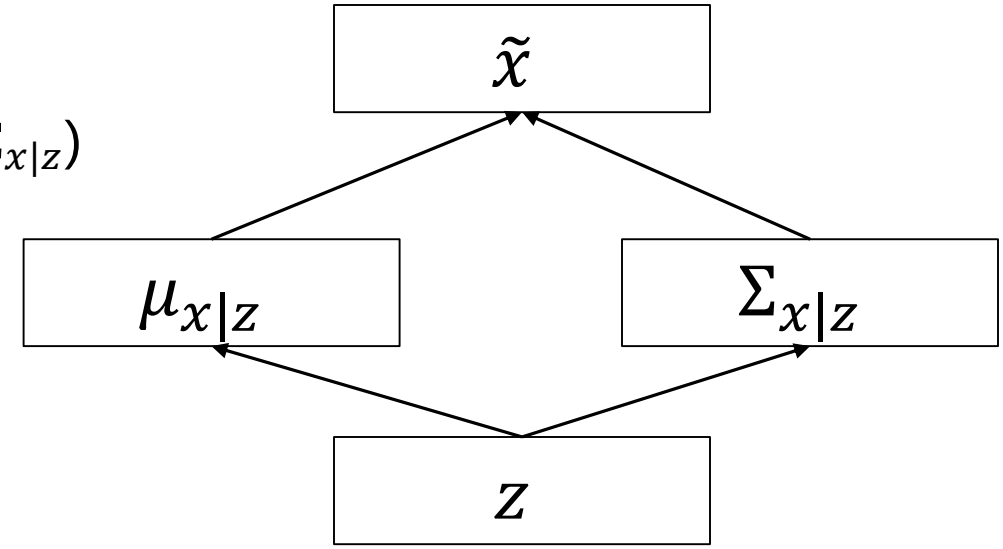
$$\begin{aligned}\mathcal{L}(q) &= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{z}, \boldsymbol{x}) + \mathcal{H}(q(\mathbf{z} \mid \boldsymbol{x})) \\ &= \mathbb{E}_{\boldsymbol{z}\sim q(\boldsymbol{z}|\boldsymbol{x})} \log p_{\text{model}}(\boldsymbol{x} \mid \boldsymbol{z}) - D_{\text{KL}}(q(\mathbf{z} \mid \boldsymbol{x}) || p_{\text{model}}(\mathbf{z})) \\ &\leq \log p_{\text{model}}(\boldsymbol{x}).\end{aligned}$$

Generating Samples from VAE

Sample x|z from $x|z \sim N(\mu_{x|z}, \Sigma_{x|z})$

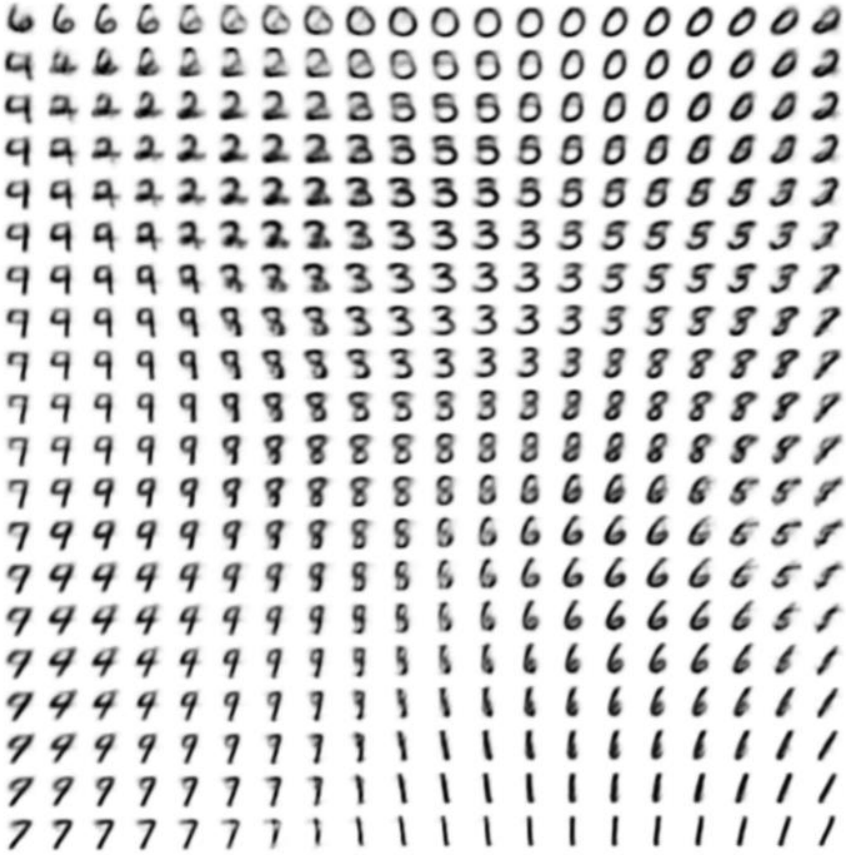
Decoder network
$p_\theta(x|z)$

Sample z from $z \sim N(0, I)$



VAE Example

(a) Learned Frey Face manifold



(b) Learned MNIST manifold

[Kingma et al., Auto-encoding Variational Bayes, 2014]

VAE: Discussion

- Strength of VAE
  - Principled approach to generative modeling
  - Allows inference of $q(z|x)$ which extracts hidden features
  - Simple to implement

- Weakness of VAE
  - Approximate inference: i.e., optimizes the lower bound ELBO
  - Samples from VAE trained on images tend to be blurry

Outline

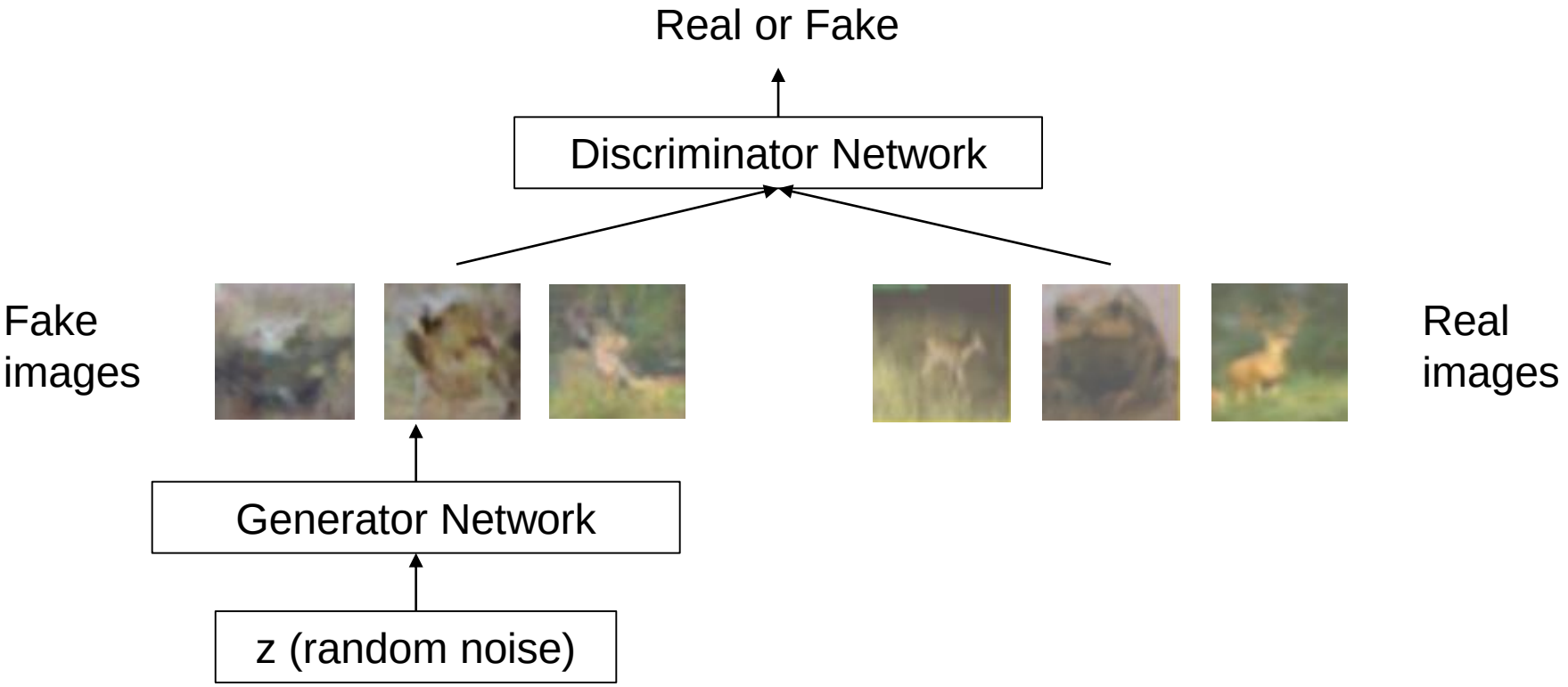
- ☑ Boltzmann Machines
- ☑ Restricted Boltzmann Machines
- ☑ Deep Belief Networks
- ☑ Deep Boltzmann Machines
- ☑ Other Boltzmann Machines
- ☑ Back-Propagation through Random Operations
- ➡ ☐ **Directed Generative Nets**
  - VAE
  - ➡ GAN

Generative Adversarial Network

- Generative Adversarial Network (GAN)
  - Another model based on differentiable generator network
  - Based on a game theoretic scenario where the generator network computes against an adversary
  - The generator network produces samples $x = g(z; \theta^{(g)})$. The adversary discriminator network attempts to distinguish samples drawn from the training data and samples from the generator
  - The discriminator emits a probability value given by $d(x; \theta^{(d)})$, the probability that x is a real training example rather than a fake sample

Generative Adversarial Network

- Generator: try to fool the discriminator
- Discriminator: try to distinguish real or fake images



Learning in GAN

- Learning is formulated as a zero-sum game, where a function $v(\theta^{(g)}, \theta^{(d)})$ determines the payoff of the discriminator. The generator receives $-v(\theta^{(g)}, \theta^{(d)})$ as its own payoff.
- During learning, each player attempts to maximize its payoff, so that at convergence
  - $g^* = \arg min_g max_d \; v(\theta^{(g)}, \theta^{(d)}) = \arg min_g max_d \; [E_{x \sim p_{data}} \log d(x) + E_{x \sim p_{generator}} \log(1 - d(x))]$
  - Note that the discriminator is a function of $\theta^{(d)}$ and the generator is a function of $\theta^{(g)}$
- This formulation enforces the discriminator to learn to correctly classify samples as real or fake. Also, the generator is trained to fool the discriminator into believing its samples are real
- At convergence, the generator's samples are indistinguishable from real data, and the discriminator outputs ½ everywhere. Then the discriminator may be discarded

Learning in GAN

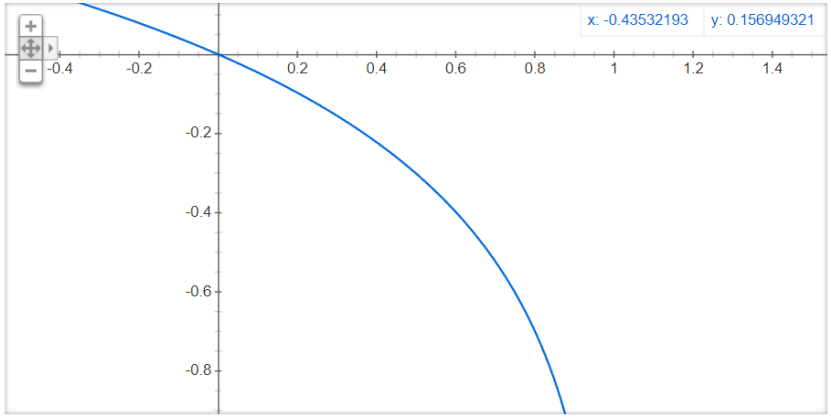
- Objective function
  - $g^* = \arg min_g max_d \; v(\theta^{(g)}, \theta^{(d)}) = \arg min_g max_d \; [E_{x \sim p_{data}} \log d(x) + E_{x \sim p_{generator}} \log(1 - d(x))]$
  - Note that the discriminator is a function of $\theta^{(d)}$ and the generator is a function of $\theta^{(g)}$

- Algorithm: alternate the following two steps
  - Step 1: update $\theta^{(d)}$ to maximize the objective $E_{x \sim p_{data}} \log d(x) + E_{x \sim p_{generator}} \log(1 - d(x)))$
  - Step 2: update $\theta^{(g)}$ to minimize the objective $E_{x \sim p_{generator}} \log(1 - d(x))$

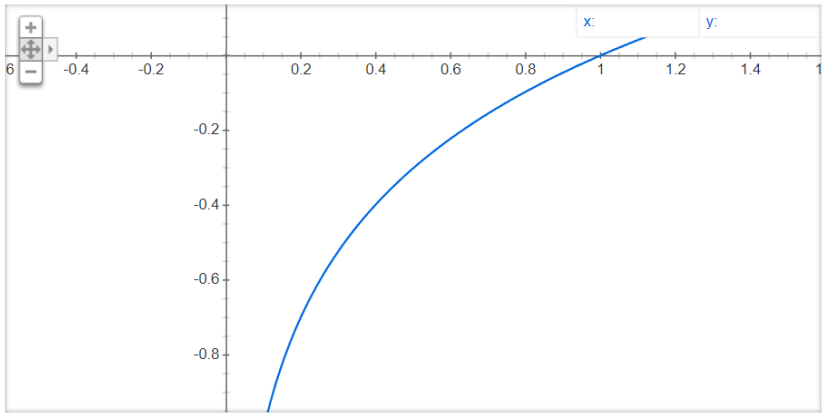
Learning in GAN

- Algorithm: alternate the following two steps
  - Step 1: update $\theta^{(d)}$ to maximize the objective $E_{x\sim p_{data}} \log d(x) + E_{x\sim p_{generator}} \log(1 - d(x)))$
  - Step 2: update $\theta^{(g)}$ to *minimize* the objective $E_{x\sim p_{generator}} \log(1 - d(x))$
- In reality: the Step 2 is reformulated as follows
  - Step 2: update $\theta^{(g)}$ to *maximize* the objective $E_{x\sim p_{generator}} \log d(x)$
  - The reason is that gradient of $\log(1 - d(x))$ is small when $d(x)$ is small, while that of $\log d(x)$ is large, so the parameters are updated quickly

Graph for log(1-x)



Graph for log(x)



Samples from GAN

a)

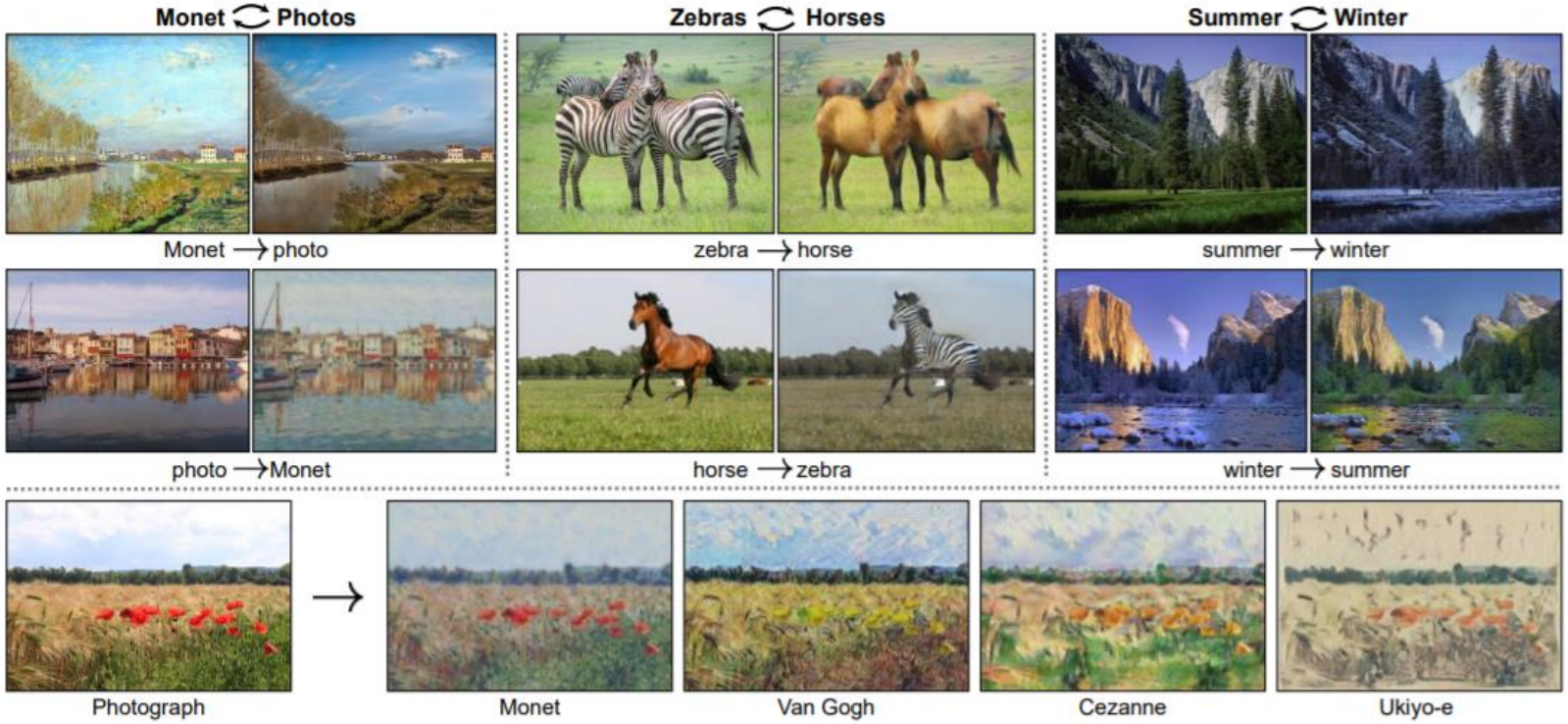
b)

c)

d)

Case Study: CycleGAN

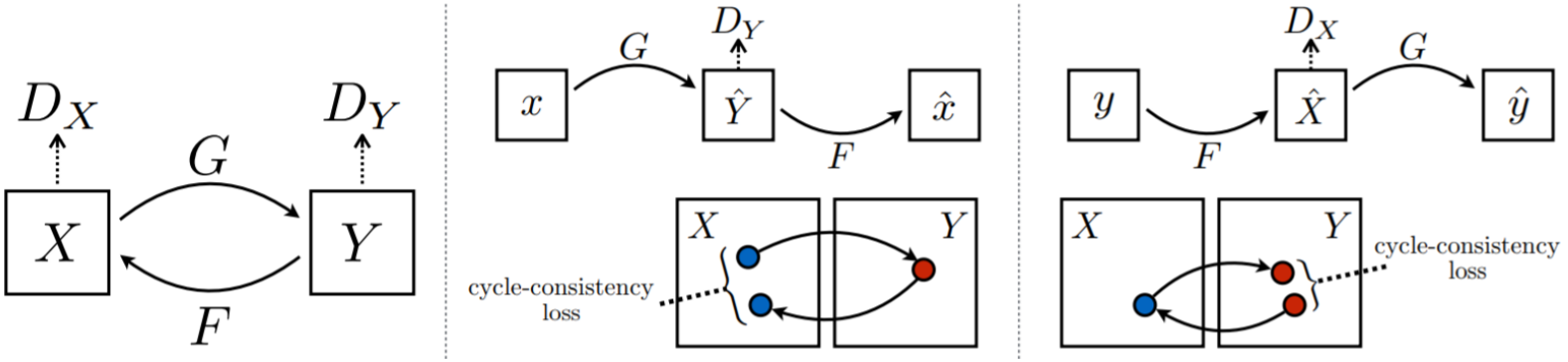
- Given any two *unordered* image collections X and Y, CycleGAN learns to automatically translate an image from one into the other and vice versa



[Zhu and Park et al., Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, 2017]

Case Study: CycleGAN

- Formulation
  - The model includes two mappings G: X -> Y, and F: Y->X
  - Two adversarial discriminators: D_X (to distinguish between images {x} and the translated images {F(y)}), and D_Y (to distinguish between images {y} and the translated images {G(x)})
  - The objective function contains two types of loss: *adversarial loss* for matching the distribution of generated images to the data distribution in the target domain, and *cycle consistency loss* to prevent the learned mappings G and F from contradicting each other



Case Study: CycleGAN

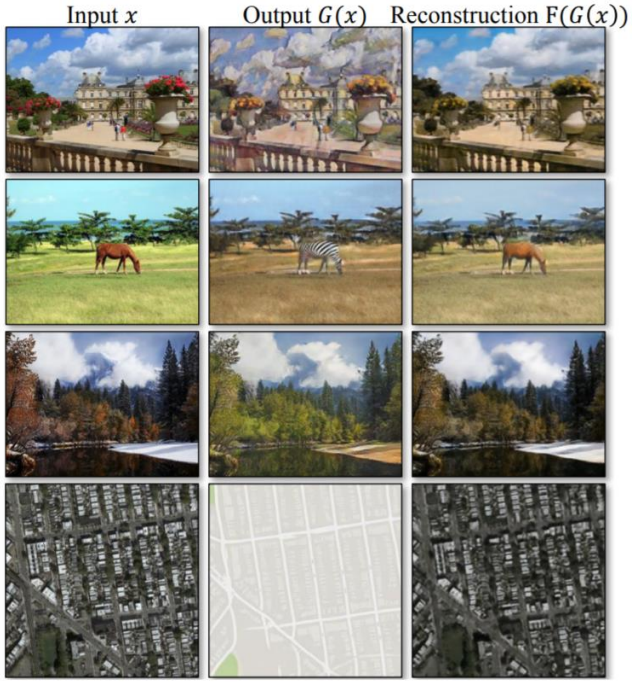
- Adversarial loss
  - For mapping function G: X -> Y and its discriminator D_Y, the objective is

$$L_{GAN}(G, D_Y, X, Y) = E_{y \sim p_{data}(y)} \log D_Y(y) + E_{x \sim p_{data}(x)} \log(1 - D_Y(G(x)))$$

  - where G tries to generate images G(x) that look similar to images from domain Y, while D_Y aims to distinguish between translated samples G(x) and real samples y
  - We want to find the best G and D_Y by optimizing the function $min_G max_{D_Y} L_{GAN}(G, D_Y, X, Y)$
  - A similar adversarial loss for the mapping function F: Y -> X and its discriminator D_X is defined as well with the objective function $min_F max_{D_X} L_{GAN}(F, D_X, Y, X)$

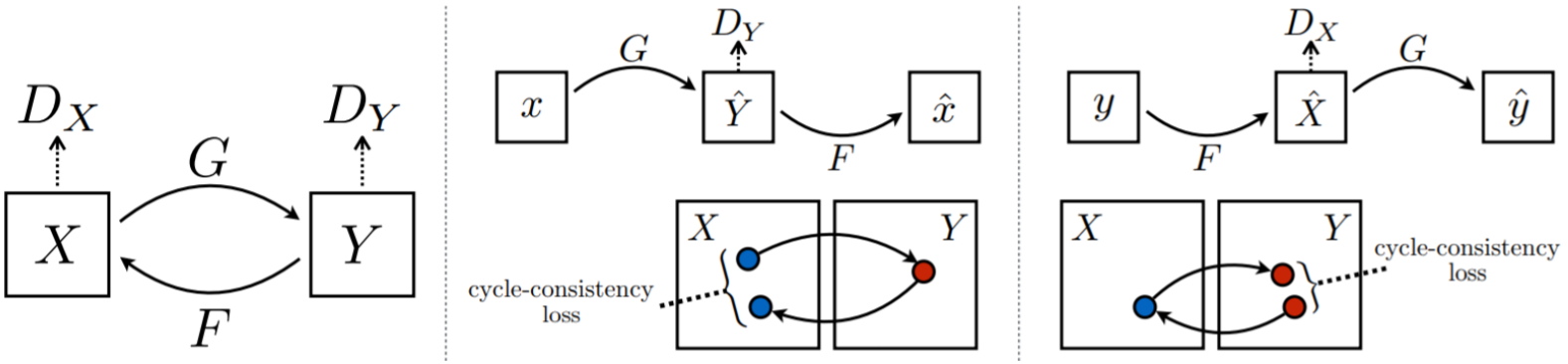
Case Study: CycleGAN

- Cycle consistency loss
  - The adversarial loss alone cannot guarantee that the learned function can map an individual input x_i to a desired output y_i. E.g., multiple images in X may be mapped to the same image in Y
  - CycleGAN introduces cycle-consistency loss
    - Forwary cycle consistency: for each image x from domain X, the image translation cycle should bring x back to the original: i.e., $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$
    - Backward cycle consistency: for each image y from domain Y, the image translation cycle should bring y back to the original: i.e., $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$
  - Cycle consistency loss: $L_{cyc}(G, F) =$ $E_{x \sim p_{data}(x)} \| F(G(x)) - x \|_1 + E_{y \sim p_{data}(y)} \| G(F(y)) - y \|_1$

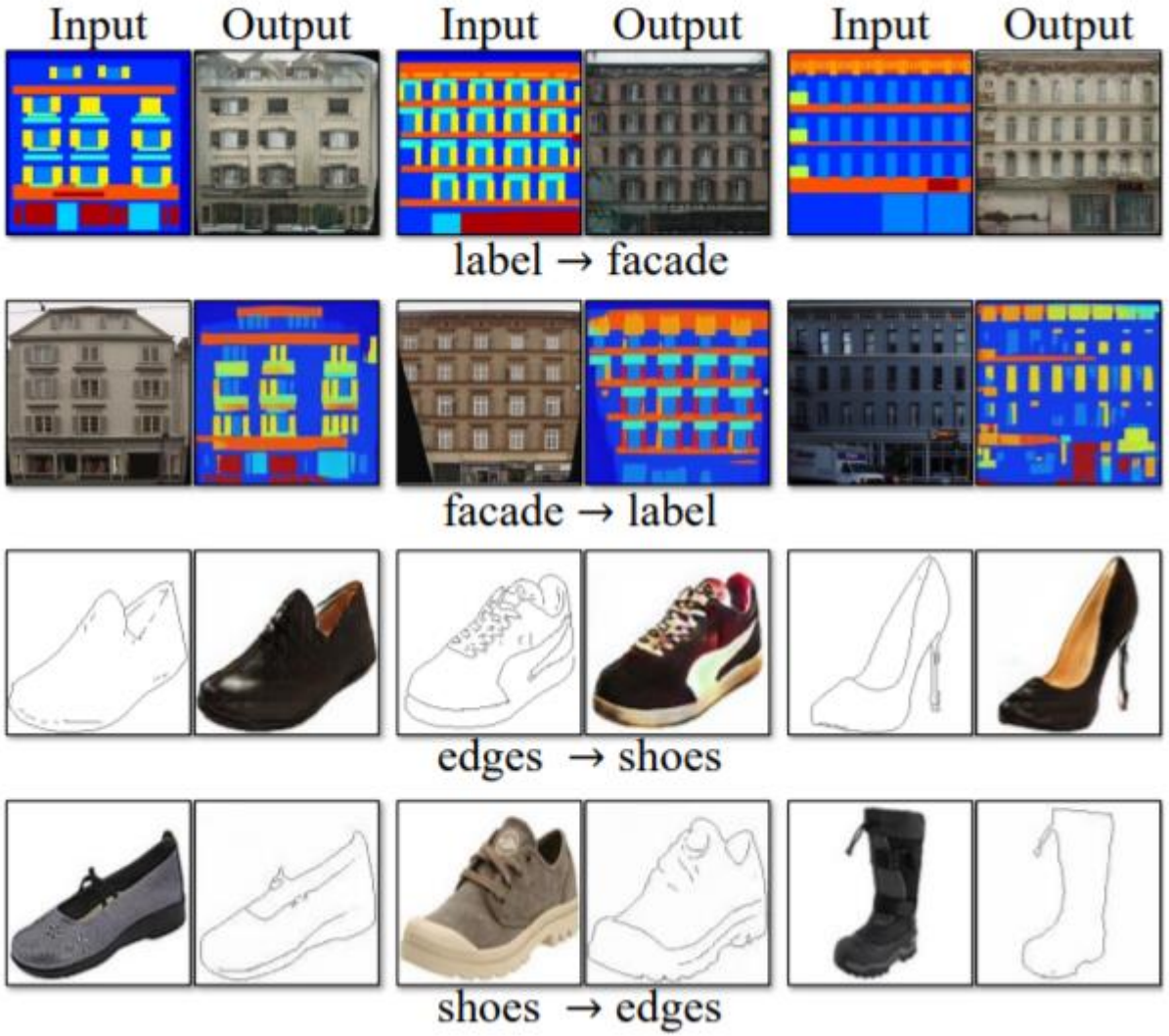


Case Study: CycleGAN

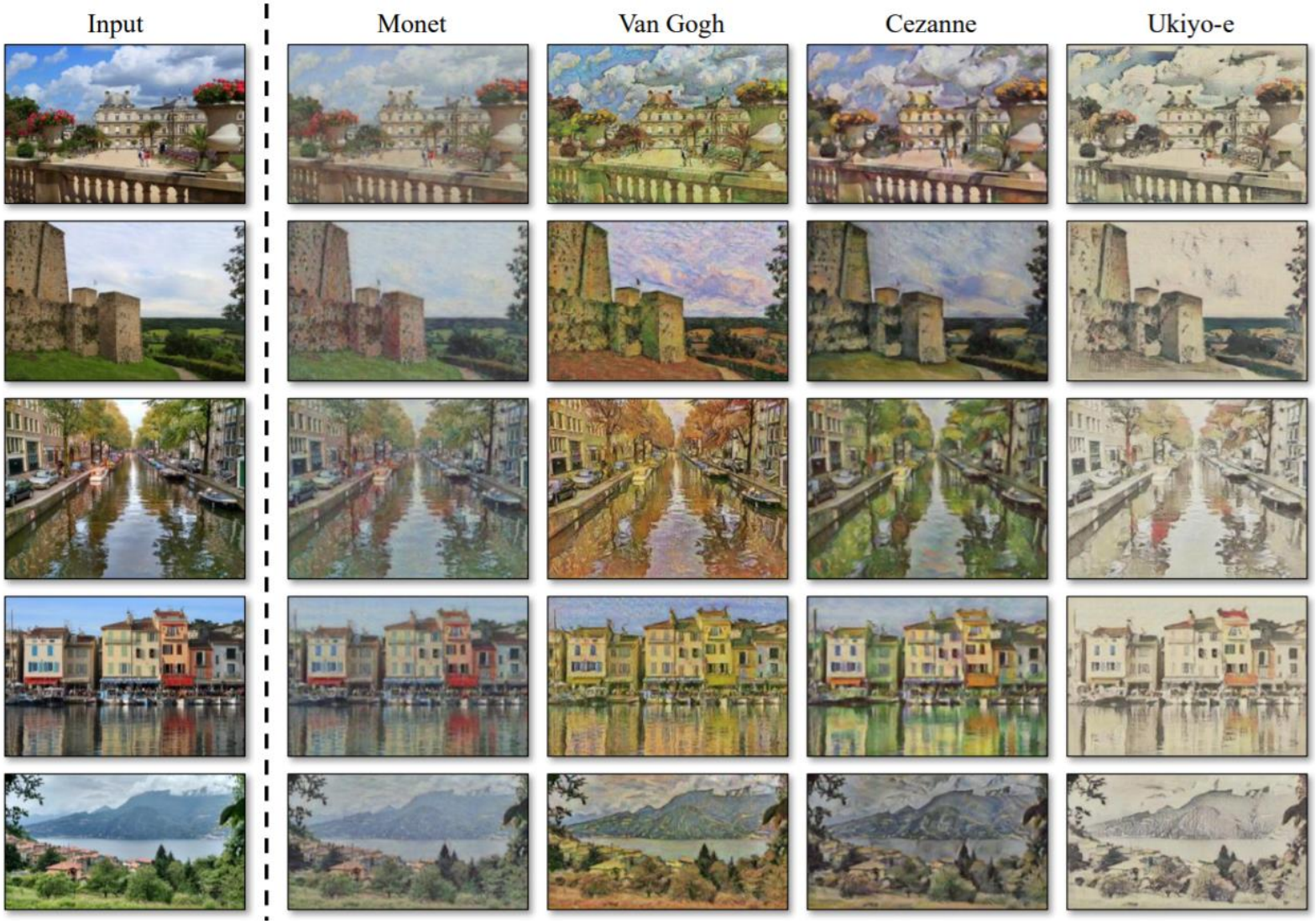
- Full objective function
  - Objective: $L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{cyc}(G, F)$
  - Goal: solve $G^*, F^* = \arg min_{G,F} max_{D_X, D_Y} L(G, F, D_X, D_Y)$



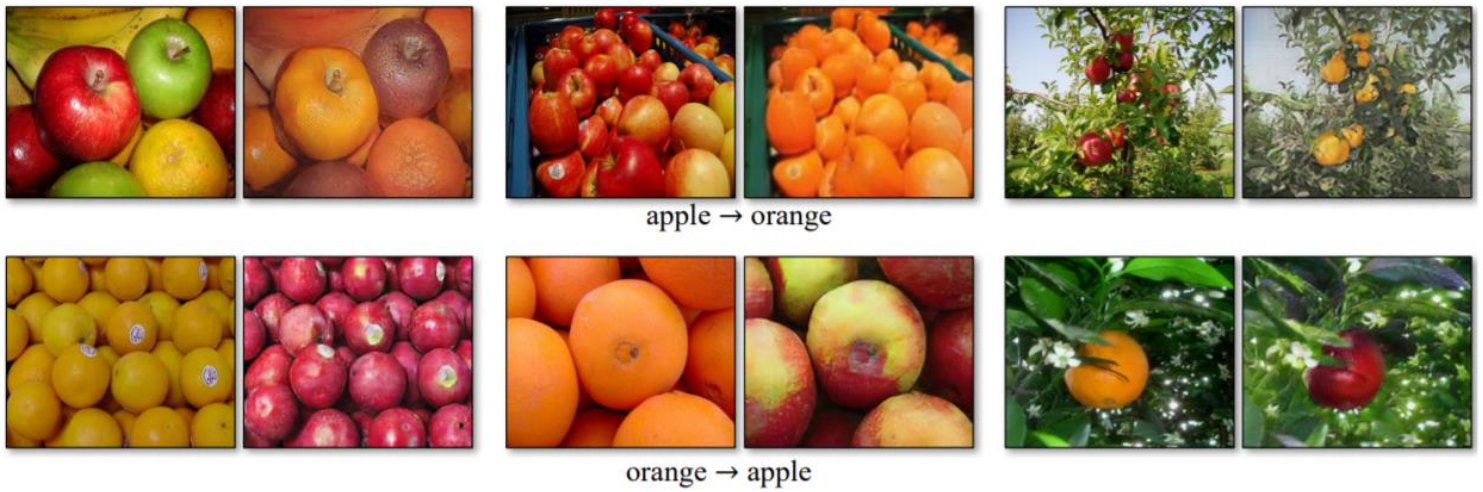
Case Study: CycleGAN

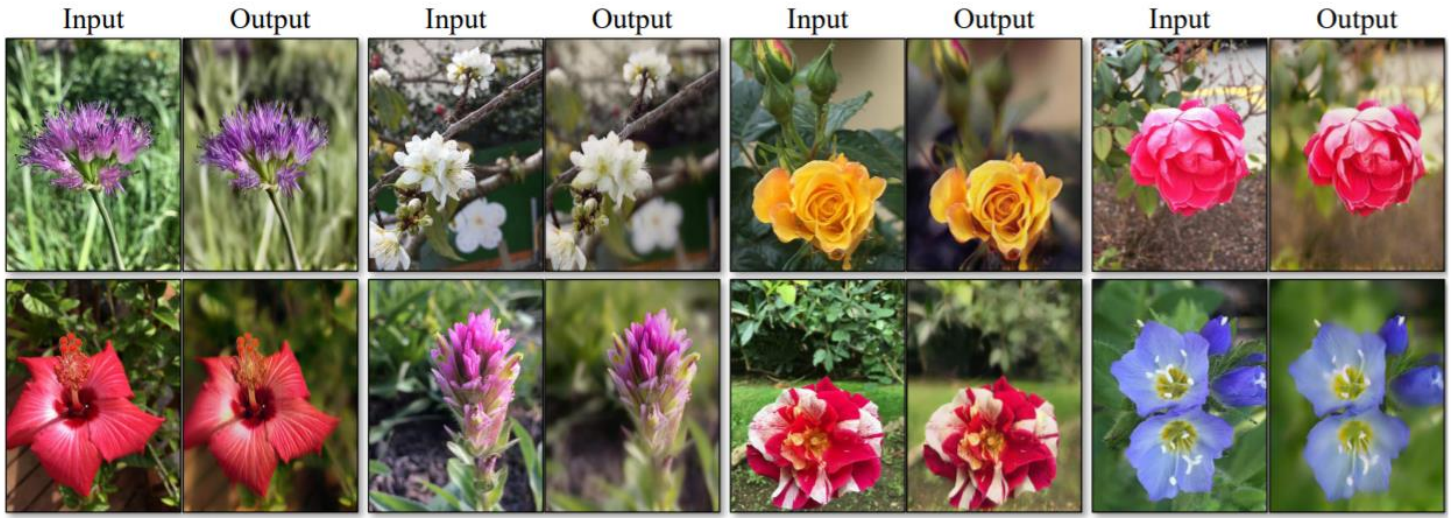


Case Study: CycleGAN



Case Study: CycleGAN





GAN: Discussion

- Strength of GAN
  - State-of-the-art image samples
- Weakness of GAN
  - Unstable to train
  - Cannot perform inference queries $q(z|x)$

What you need to know

- Inference as Optimization
- Expectation Maximization
- MAP Inference and Sparse Coding
- Variational Inference and Learning

Questions?