Bios 740- Chapter 6. Generative Adversarial Networks (GAN)

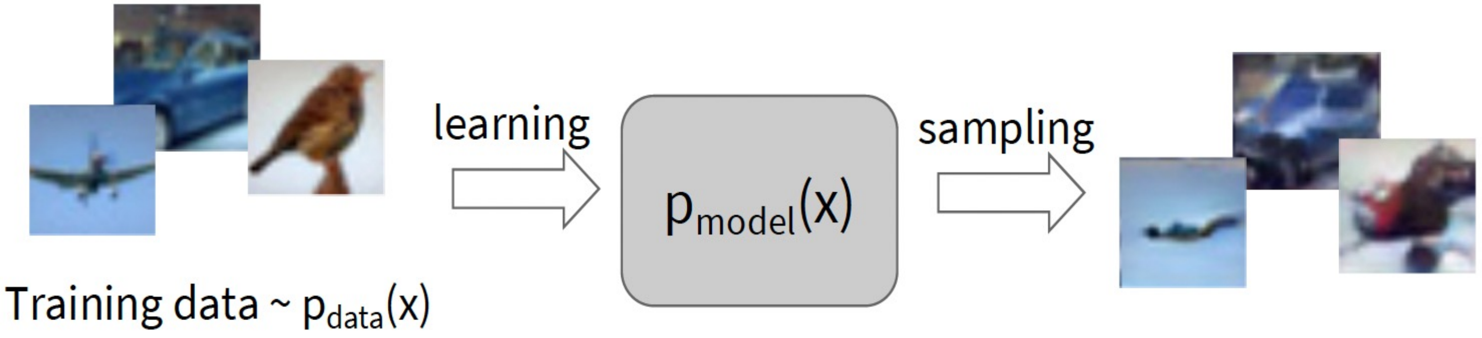
Acknowledgement: Many thanks to Mr. Xiaoqi Liu for preparing some of these slides, and to Drs. Mingxia Liu, Gang Li, Xiao Wang, and Jian Huang for sharing their slides. I also drew on material from the lecture presentations of Stanford CS224w and StanfordCS231n, Dr. Liu's presentation, and Dr. Huang's lecture, as well as content generated by ChatGPT.

Content

Content

What are Generative Models?

Definition: Generative models learn to generate new data samples resembling a given dataset.



Objectives:

1. Learn $p_{model}(x)$ that approximates $p_{data}(x)$
2. Sampling new x from $p_{model}(x)$

Major Generative Models:

- **Explicit Density Models:** Estimate probability distributions (e.g., Gaussian Mixture Models, VAEs).
- **Implicit Density Models:** Generate samples without explicit density estimation (e.g., Generative Adversarial Network (GAN)s, Diffusion Models).

Explicitly Density Models

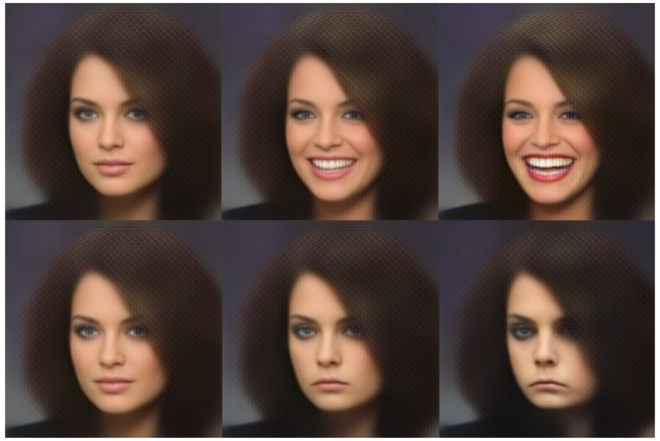
A simple form of generative learning is to learn a deterministic function

- Generate $\eta \sim N(\mathbf{0}, \mathbf{I}_m)$, $m \geq 1$.
- Estimate a generator function G such that

$$G(\eta) \sim P_X.$$

- Let $X = G_\theta(Z)$ with $Z \sim p_Z$. The distribution function of X is

$$P(X \leq \boldsymbol{x}) = P(G_\theta(Z) \leq \boldsymbol{x}) = \int_{G_\theta(\boldsymbol{z}) \leq \boldsymbol{x}} p_Z(\boldsymbol{z}) d\boldsymbol{z}$$



with density function

$$p_{G_\theta}(\boldsymbol{x}) = \frac{\partial}{\partial \boldsymbol{x}} \int_{G_\theta(\boldsymbol{z}) \leq x} p_Z(\boldsymbol{z}) dz.$$

If $G_\theta(\boldsymbol{z})$ is invertible, then we have

$$p_{G_\theta}(\boldsymbol{x}) = \pi(G_\theta^{-1}(\boldsymbol{x})) |\det(\nabla_x G_\theta^{-1}(\boldsymbol{x}))|.$$

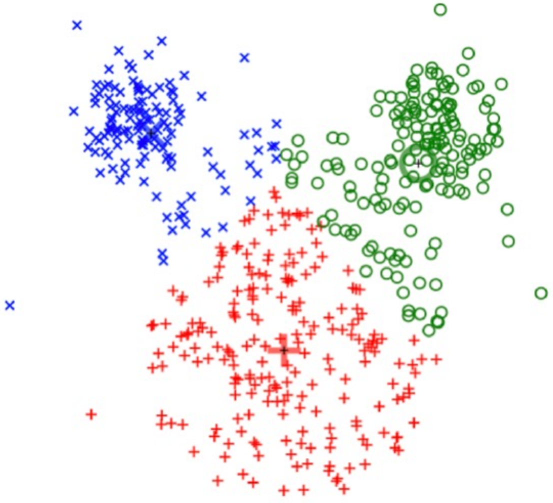
How?

It is still difficult to calculate the inverse G_θ^{-1}.

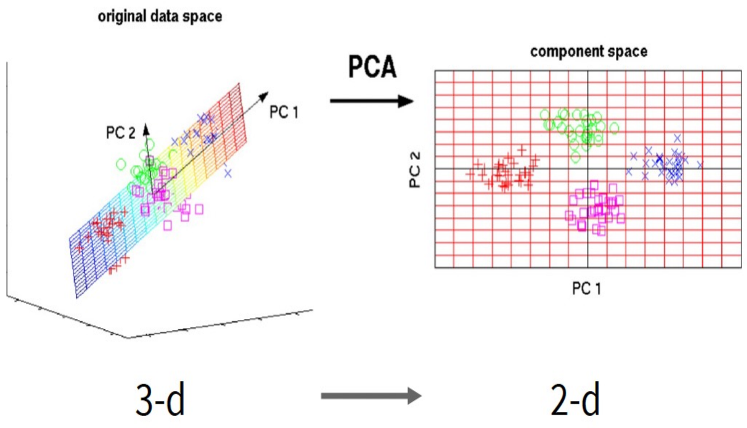
Unsupervised Learning

- **Data:** X — Just data, no labels
- **Goal:** Learn some underlying hidden structure or distribution of the data
- **Examples:** clustering, dimension reduction, feature learning, density estimation, etc.

- **Generative models are a subset of unsupervised learning, but not all unsupervised learning techniques are generative (e.g., k-means, PCA)**



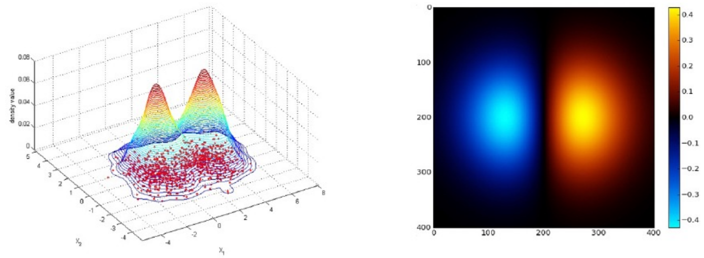
K-means clustering



PCA

Figure copyright Ian Goodfellow, 2016. Reproduced with permission.

1-d density estimation



2-d density estimation

Modeling p(x)

2-d density images left and right are CC0 public domain

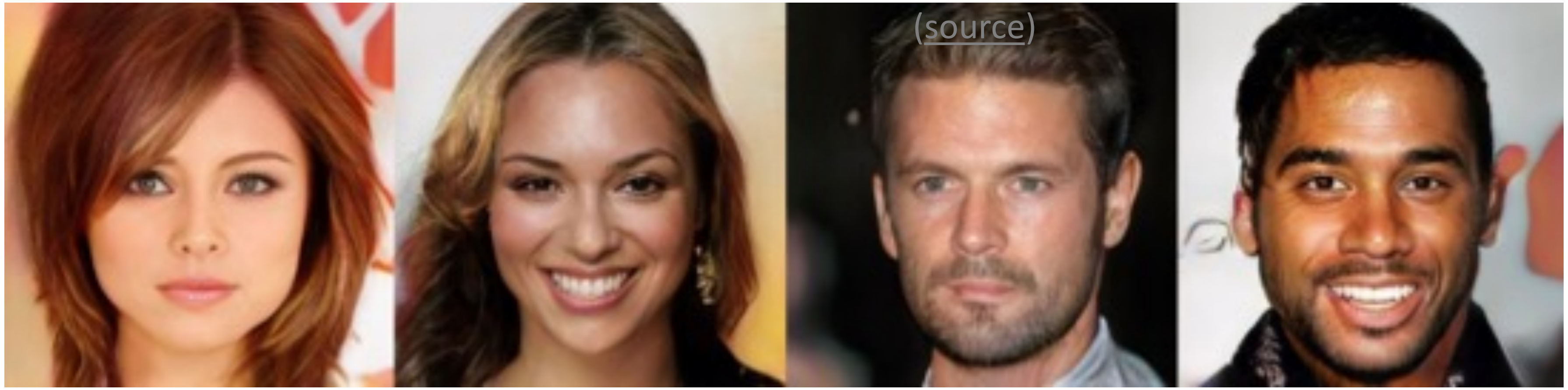
Why GANs and Other Generative Models?

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative adversarial nets." *Advances in neural information processing systems* 27 (2014).

GANs Progression on Face Generation

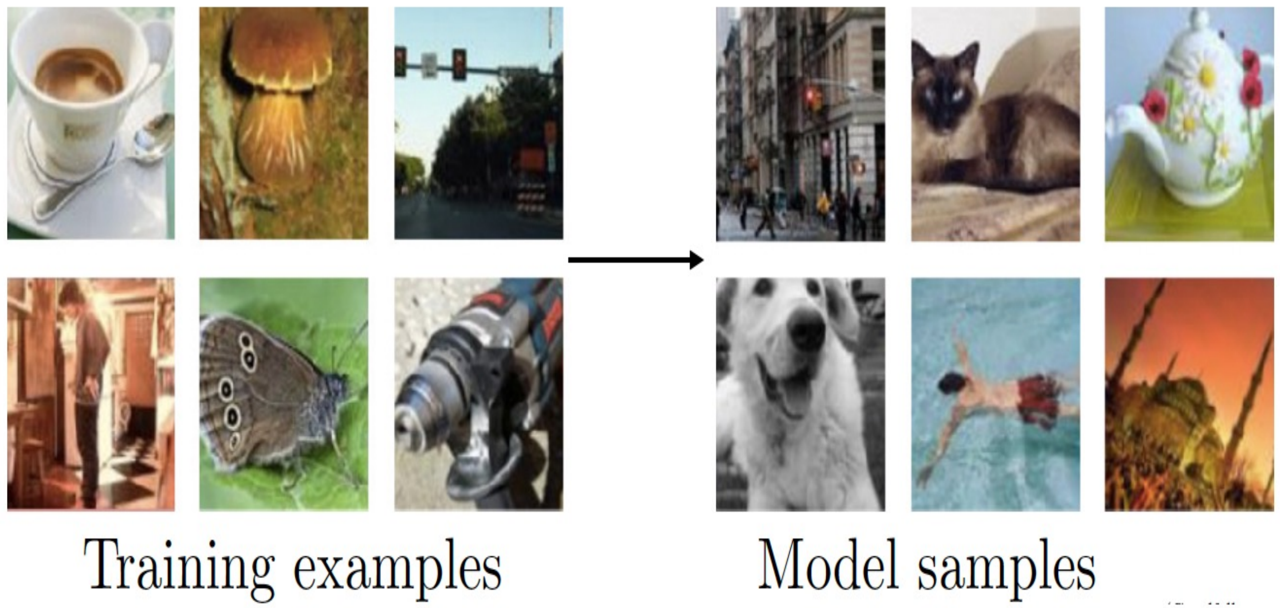
- Better Quality
- High Resolution

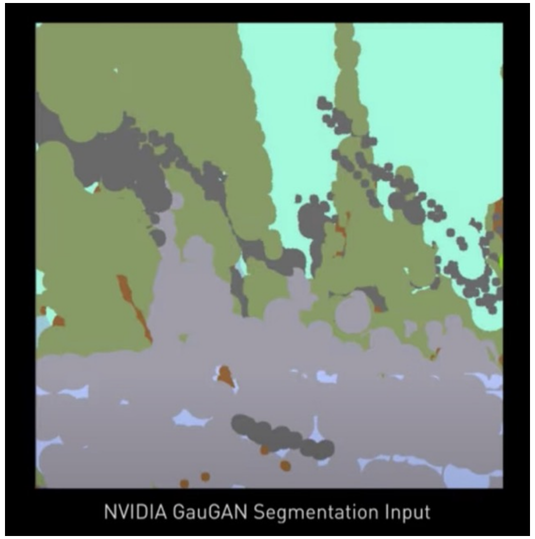
(source)



1024*1024 Images generated by a GAN created by NVIDIA. (source, 2018)

Why Gans?

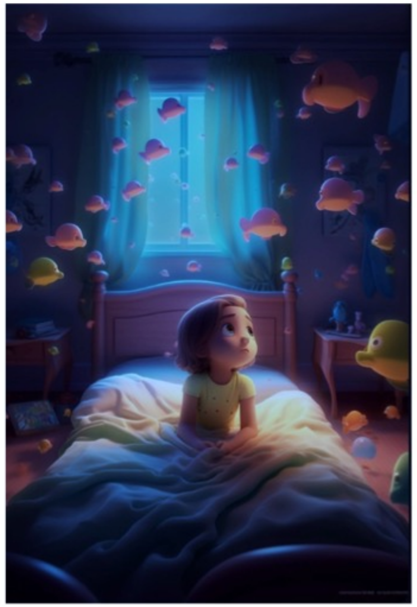




Realistic samples for artwork, super-resolution, colorization, etc.

- Learn useful and subtle features for downstream tasks such as classification and object detection.
- Getting insights from high-dimensional data (physics, medical imaging, etc.)
- Modeling physical world for simulation and planning (robotics and reinforcement learning applications)
- Many more ...

AI Art



Midjourney v5

Emerging Generative Models in 2022-

Imagen

PHENAKI

alphacode
it solutions

Parti
Pathways Autoregressive Text-to-Image Model

DESIGNS.AI

OpenAI
ChatGPT

DALL·E 2

Stable Diffusion

What are Conditional Generative Models?

Definition: A **conditional generative model** is a type of generative model that generates new data samples based on additional context or conditioning information. Mathematically, it models the conditional probability distribution,

Applications: Image-to-Image translation, Text-to-Image generation, Super-Resolution, etc.

$$\boxed{P(x \mid y) = \frac{P(y \mid x)}{P(y)} P(x)}$$

$$\underbrace{\boxed{P(x \mid y)}}_{\text{Conditional Generative Model}} = \frac{\overbrace{\boxed{P(y \mid x)}}^{\text{Discriminative Model}}}{\underbrace{\boxed{P(y)}}_{\text{Prior over labels}}} \overbrace{\boxed{P(x)}}^{\text{(Unconditional) Generative Model}}$$

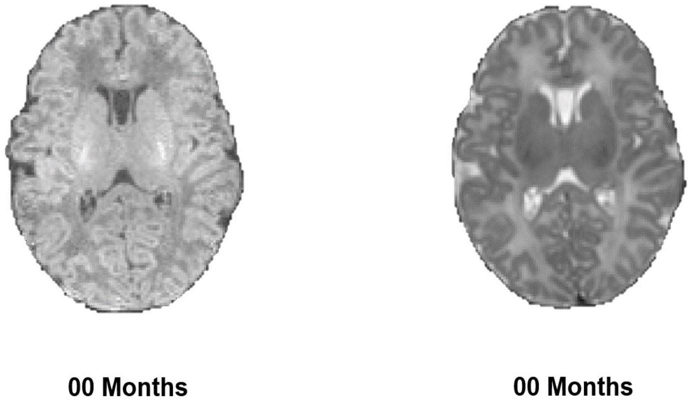
We can build a conditional generative model from other components!

Why Conditional Generative Models?

Outfilling/Missing Data Imputation

← Generated Input Generated →

T1w<---------→T2w



- **Infant brain MR images (T1w/T2w)**
  - **Low** tissue contrast and **dynamic** change in appearance

Additional Applications

Text to Image

Fashion Design

Content

Deep Generative Models

Deep generative models are neural network-based models designed to learn complex data distributions and generate realistic synthetic samples that resemble the original training data. These models leverage deep learning to approximate the true underlying data distribution.

Let $X \sim P_X$, where P_X is the distribution of X. Let its density function be p_X. There are two ways to learn the distribution of X:

- The explicit modeling approach assumes $p_X \in \mathcal{P}_\Theta$, or estimates p_X directly nonparamtrically.
- Generative models learn a generator function $G : \mathbb{R}^m \to \mathbb{R}^p$ such that $G(\eta) \sim P_X$, where $\eta \sim P_\eta$, a known reference distribution.
  - If a generator function G is known, then we know everything about P_X, since we can first sample $\eta \sim P_\eta$, then $G(\eta) \sim P_X$.
  - We usually take the reference distribution to be $N(\mathbf{0}, \mathbf{I}_m)$ or uniform distribution on $[0, 1]^m$.

Common DGMs:

- Deep Belief Networks (DBNs)
- Deep Boltzmann Machines (DBMs)
- Denoising Autoencoders (DAEs)
- Generative Stochastic Networks (GSNs)
- PixelRNN/PixelCNN
- Generative Adversarial Network (GANs)
- Variational Autoencoder

Taxonomy of Deep Generative Models

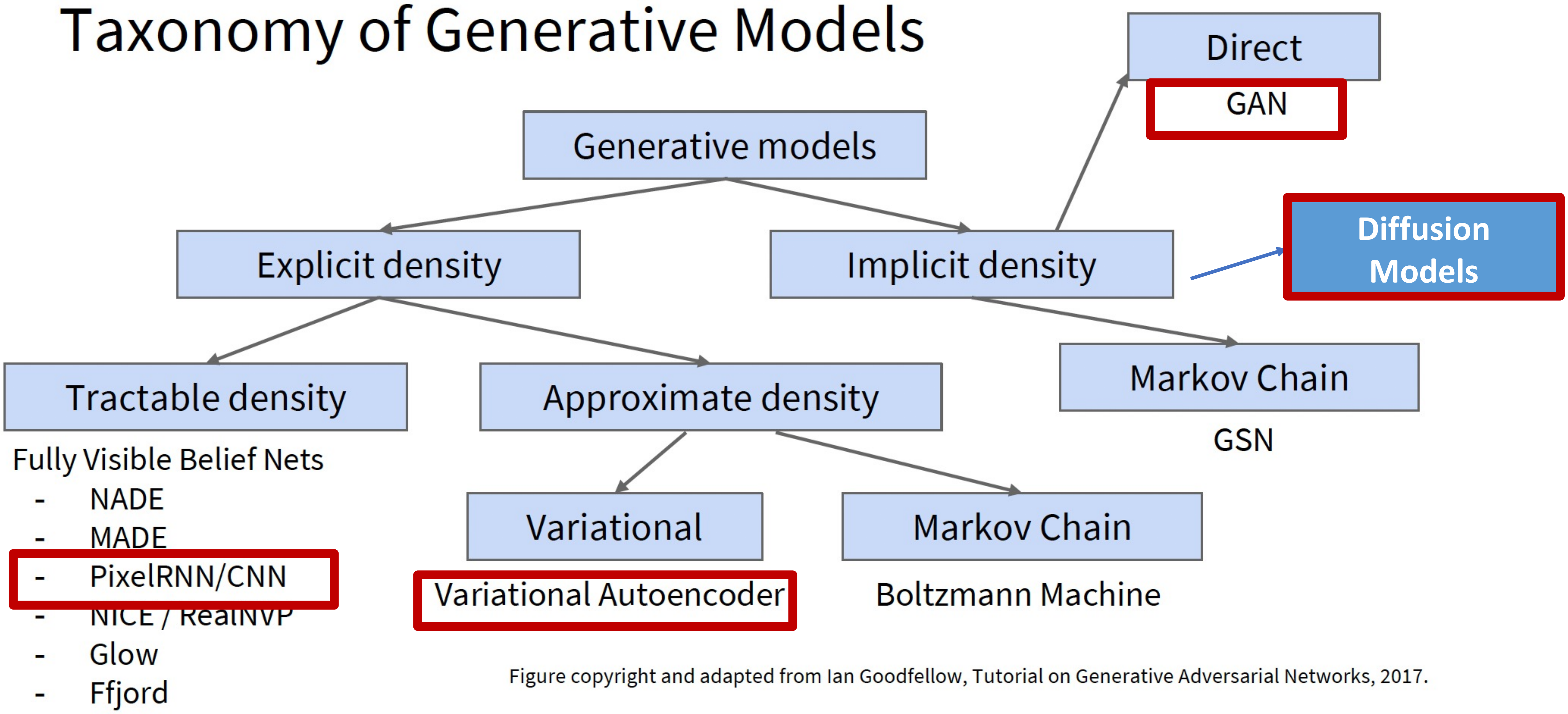
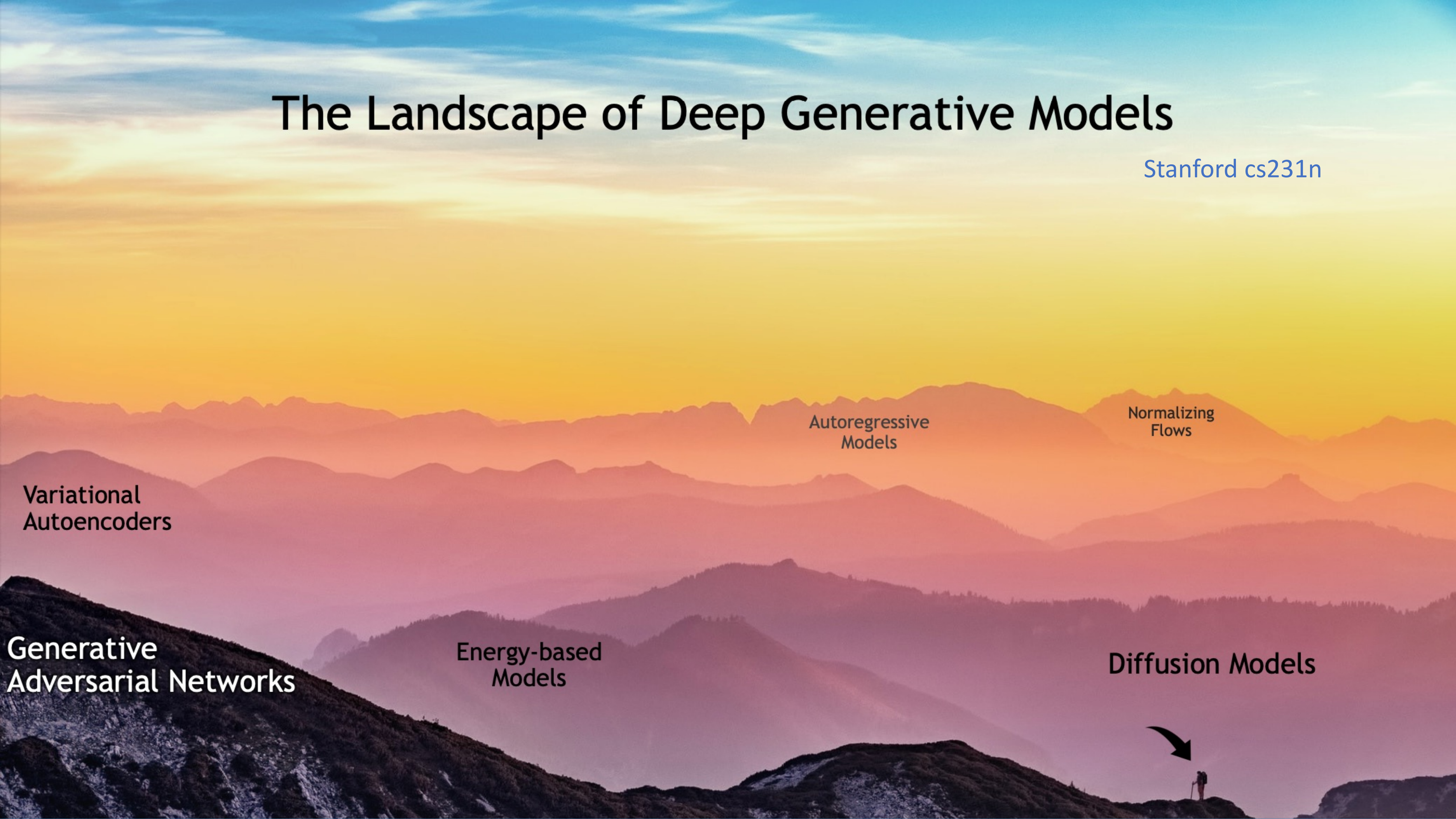


Figure copyright and adapted from Ian Goodfellow, Tutorial on Generative Adversarial Networks, 2017.

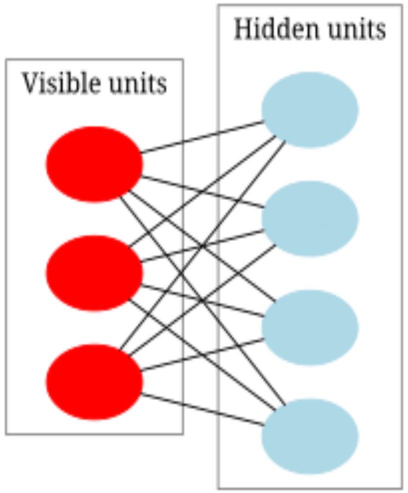
The Landscape of Deep Generative Models

Stanford cs231n



DBN and DBM

Restricted Boltzmann Machines (RBMs)



- **Energy Function:**

$$E(v,h) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{i,j} v_i W_{ij} h_j$$

- **Probability Distribution:**

$$P(v,h) = \frac{e^{-E(v,h)}}{Z}, \quad Z = \sum_{v,h} e^{-E(v,h)}$$

$$P(v) = \frac{1}{Z} \sum_{\{h\}} e^{-E(v,h)}$$

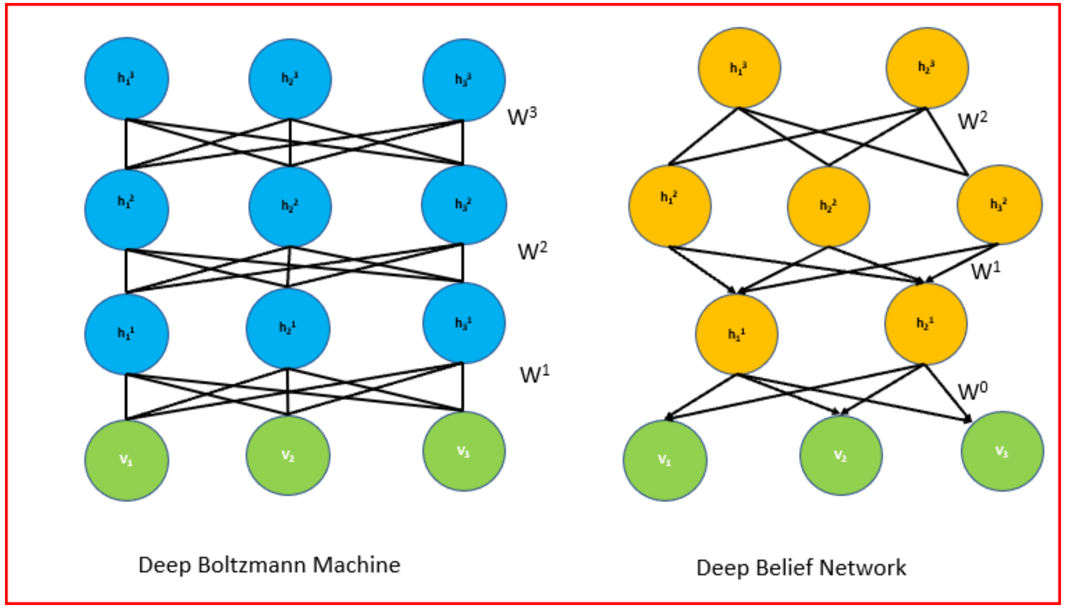
DBNs stack multiple RBMs to form a deep probabilistic model.

$$P(v, h^1, h^2, ..., h^L) = P(h^L) \prod_{k=1}^{L-1} P(h^k | h^{k+1}) P(v | h^1)$$

DBMs extend RBMs into a multi-layer structure, allowing deeper hierarchical representations.

$$E(v, h^1, h^2) = -\sum_i a_i v_i - \sum_j b_j h_j^1 - \sum_k c_k h_k^2$$

$$-\sum_{ij} v_i W_{ij} h_j^1 - \sum_{jk} h_j^1 U_{jk} h_k^2$$



Denoising Autoencoders (DAE)

A variant of autoencoders that learns robust representations by reconstructing clean inputs from noisy versions.

- ❖ The goal is to force the network to capture meaningful structure while ignoring noise.
- ❖ Used in image denoising, feature learning, and semi-supervised learning.

- ▶ A corrupted version is generated as $\tilde{\mathbf{x}} \sim q_D(\tilde{\mathbf{x}}|\mathbf{x})$, where noise is added.
- ▶ The encoder maps the noisy input to a latent representation:

$$\mathbf{h} = f_\theta(\tilde{\mathbf{x}}) = \sigma(W\tilde{\mathbf{x}} + b)$$

- ▶ The decoder reconstructs the original input:

$$\mathbf{r} = g_\phi(\mathbf{h}) = \sigma(W'\mathbf{h} + b')$$

- ▶ **Noise Injection:** The corruption process $q_D(\tilde{\mathbf{x}}|\mathbf{x})$ can be:
  - ▶ Gaussian noise: $\tilde{\mathbf{x}} = \mathbf{x} + \mathcal{N}(0, \sigma^2 I)$.
  - ▶ Masking: Randomly setting input values to zero.
  - ▶ Salt-and-pepper noise.
- ▶ The reconstruction error is minimized:

$$\mathcal{L}(\theta, \phi) = \mathbb{E}_{\mathbf{x}\sim p_{data}, \tilde{\mathbf{x}}\sim q_D}\left[L(\mathbf{x}, g_\phi(f_\theta(\tilde{\mathbf{x}})))\right]$$

where $L(\mathbf{x}, \mathbf{r}) = \|\mathbf{x} - \mathbf{r}\|_2^2$.

- ▶ DAEs generate samples by reconstructing corrupted inputs:

$$\tilde{X} \sim q(\tilde{X}|X), \quad h = f_\theta(\tilde{X}), \quad \hat{X} = g_\phi(h)$$

Generative Stochastic Networks (GSNs)

•**Definition**: GSNs are a class of generative models that learn to transform noise into structured data by learning a Markov transition function. GSNs sidestep intractable partition functions.

- ❖ They generalize Denoising Autoencoders (DAEs) by introducing latent variables into the learning process.
- ❖ Training involves learning a transition function that converges to the data distribution

GSNs use the denoising autoencoder to define a **Markov chain** that generates samples. The Markov chain is defined by a transition operator $T(x' \mid x)$.

- Let x_t be the state of the Markov chain at step t
- The next state x_{t+1} is generated by:
  1. Corrupting x_t with noise: $\tilde{x}_t \sim q(\tilde{x} \mid x_t)$.
  2. Applying the denoising autoencoder to obtain $x_{t+1} = f_\theta(\tilde{x}_t)$.
- The transition operator $T(x_{t+1} \mid x_t)$ can be written as:

$$T(x_{t+1} \mid x_t) = \int q(\tilde{x} \mid x_t)\, \delta(x_{t+1} - f_\theta(\tilde{x}))\, d\tilde{x}$$

where $\delta(\cdot)$ is the Dirac delta function.

Iterative Markov chain with noise injection:

$$X_{t+1} = f_\theta(X_t, Z_t), \quad Z \sim p(Z)$$

Generative Stochastic Networks (GSNs)

To generate new samples from the GSN:

1. Start with an initial state x_0 (e.g., random noise).
2. Iteratively apply the transition operator $T(x_{t+1} \mid x_t)$ for T steps:
$$x_{t+1} = f_\theta(\tilde{x}_t), \quad \tilde{x}_t \sim q(\tilde{x} \mid x_t)$$
3. After sufficient steps, x_T will be a sample from $p_{\text{model}}(x)$.

$$\theta^* = \arg\min_\theta \mathbb{E}_{x\sim p_{\text{data}},\tilde{x}\sim q(\tilde{x}|x)} \left[||x - f_\theta(\tilde{x})||^2 \right]$$

- p_{data} represents the true data distribution.
- $q(\tilde{x}|x)$ defines the corruption process introducing noise.
- $f_\theta(\tilde{x})$ is the model that reconstructs x from corrupted input $\tilde{x}$.

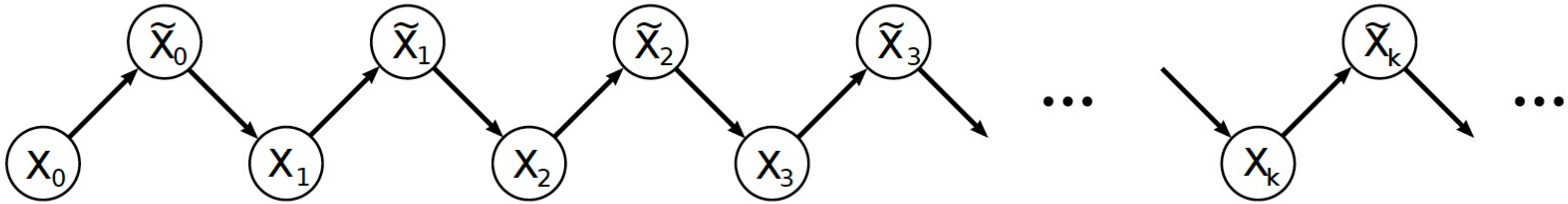
The training objective of GSNs is to minimize the reconstruction error of a denoising autoencoder while ensuring that the Markov chain converges to the desired distribution. Additionally, regularization techniques or constraints can be applied to stabilize the Markov chain during training.

Generative Stochastic Networks (GSNs)

Proposition 2 *Let $P(X)$ be the training distribution for which we only have empirical samples. Let $\pi(X)$ be the implicitly defined asymptotic distribution of the Markov chain alternating sampling from $P_\theta(X|\tilde{X})$ and $\mathcal{C}(\tilde{X}|X)$, where $\mathcal{C}$ is the original local corruption process.*

If we assume that $P_\theta(X|\tilde{X})$ has sufficient capacity and that the walkback algorithm converges (in terms of being stable in the updates to $P_\theta(X|\tilde{X})$), then $\pi(x) = P(X)$.

That is, the Markov chain defined by alternating $P_\theta(X|\tilde{X})$ and $\mathcal{C}(\tilde{X}|X)$ gives us samples that are drawn from the same distribution as the training data.



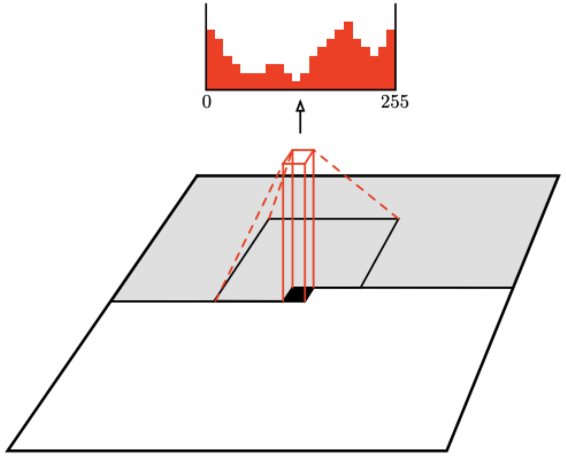
Alain, Guillaume, Yoshua Bengio, Li Yao, Jason Yosinski, Eric Thibodeau-Laufer, Saizheng Zhang, and Pascal Vincent. "GSNs: generative stochastic networks." *Information and Inference: A Journal of the IMA* 5, no. 2 (2016): 210-249.

PixelRNN/PixelCNN

Autoregressive models predict the probability of an image as a sequential product of pixel conditionals:

- ❖ Each pixel is generated sequentially, conditioned on all previously generated pixels.
- ❖ PixelRNN and PixelCNN differ in how they model these dependencies.

$$p(\mathbf{x}) = \prod_{i=1}^{N} p(x_i | x_1, x_2, \ldots, x_{i-1})$$



1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

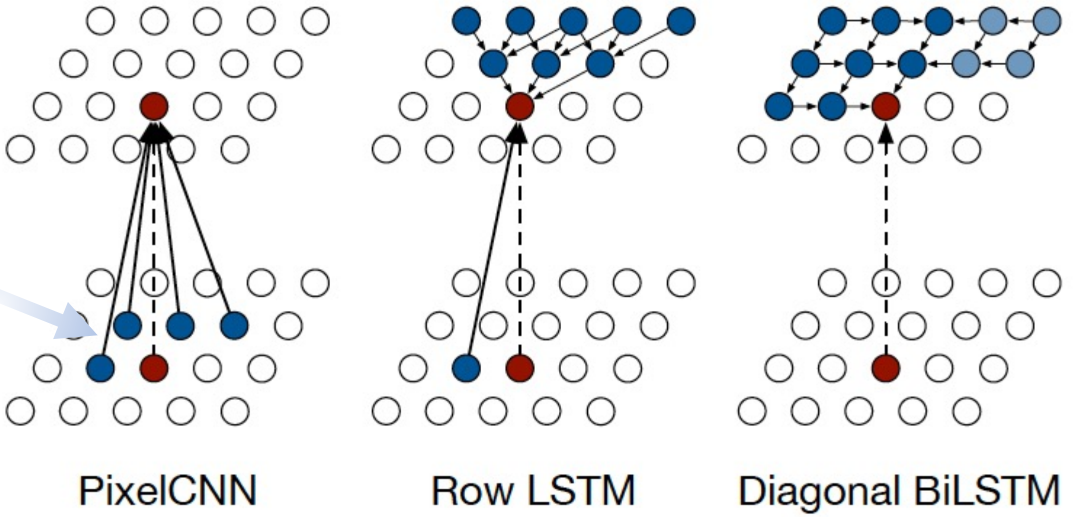
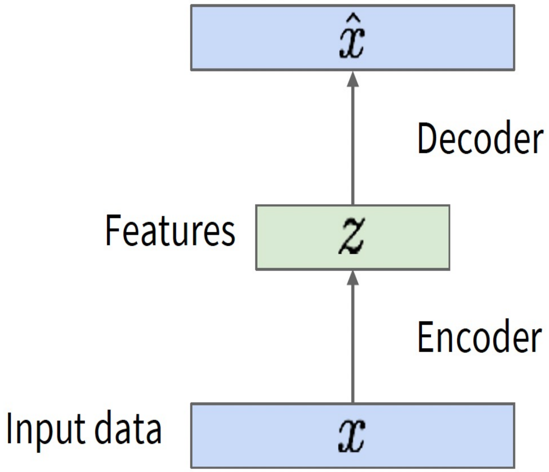


Figure 4. Visualization of the input-to-state and state-to-state mappings for the three proposed architectures.

Variational Autoencoder

Key Idea:

- Encoder: $q(z \mid x) \approx$ posterior.
- Decoder: $p(x \mid z)$.
- Prior: $p(z) = \mathcal{N}(0, I)$.



Objective (ELBO):

$$\mathcal{L}_{\text{VAE}} = \mathbb{E}_{q(z|x)}[\log p(x \mid z)] - \text{KL}(q(z \mid x) \| p(z))$$

Training:

- Maximize ELBO using stochastic gradient descent.

Encoder/Decoder:

$$q(z \mid x) = \mathcal{N}(z; \mu(x), \sigma^2(x)), \quad p(x \mid z) = \mathcal{N}(x; \mu(z), \sigma^2(z))$$

KL Divergence:

$$\text{KL}(q(z \mid x) \| p(z)) = \frac{1}{2}\left(\text{tr}(\sigma^2(x)) + \|\mu(x)\|^2 - d - \log\det(\sigma^2(x))\right)$$

Reparameterization Trick:

$$z = \mu(x) + \sigma(x) \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

Generative Adversarial Network (GANs)

Key Idea:

- Two networks compete:
  - Generator G: Maps noise $z \sim p_z(z)$ to $G(z)$.
  - Discriminator D: Outputs $D(x) \in [0, 1]$.

Objective:

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

Training:

1. Update D to maximize $V(D, G)$.
2. Update G to minimize $V(D, G)$.

Comparisons among DGMs

Model	Quality	Diversity	Pros	Cons
DBN	Moderate	Moderate	Efficient pretraining, useful for feature extraction	Not a true generative model, lacks flexibility
DBM	High	High	Captures complex dependencies, deep representation learning	Hard to train, requires MCMC sampling
DAE	Moderate	Low	Effective for representation learning, robust to corruption	Not a true generative model, lacks explicit sampling mechanism
GSN	High	High	No intractable partition function, stable training	Requires well-tuned Markov transitions
PixelRNN / PixelCNN	Very High	Low	High-quality samples, avoids adversarial training issues	Slow sampling (PixelRNN), limited long-range dependencies (PixelCNN)
GAN	Very High	Low	Generates highly realistic images, fast sampling	Mode collapse, unstable training
VAE	Moderate	High	Well-defined latent space, meaningful representations	Blurry image samples, over-regularized latent space

- GANs and Pixel-based models achieve the highest quality but can struggle with diversity.
- VAEs and GSNs offer high diversity but often produce lower quality outputs.
- DBMs and GSNs allow complex feature learning but are harder to train than DBNs.
- PixelCNN is faster and more stable than PixelRNN due to parallel processing.
- DAEs and VAEs are better for representation learning but are not ideal for direct sample generation.

Evaluation Metrics

Metric	Measures	Best for	Limitations
Inception Score (IS)	Quality & diversity	Image GANs	Doesn't compare to real data
Fréchet Inception Distance (FID)	Realism & diversity	Image GANs	Requires feature extraction
Precision & Recall	Fidelity & coverage	Any model	Computationally expensive
Log-Likelihood	Probability assignment	VAEs, Flows	Doesn't match human perception
Human Evaluation	Subjective quality	Any model	Expensive and subjective
Downstream Task Performance	Utility in real tasks	Task-driven models	Domain-dependent

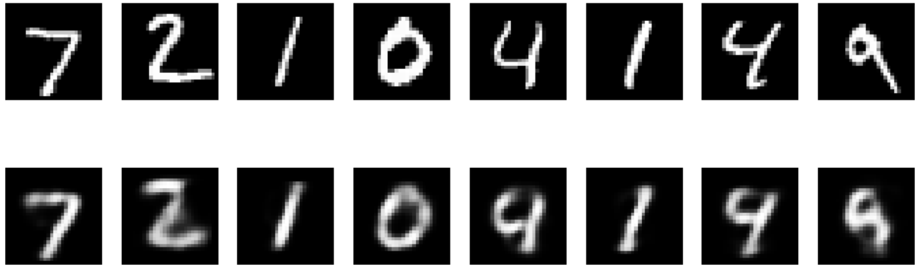
Applications of Generative Models in AI

- **Understanding Probability Distributions**
  - Generative models help represent and manipulate high-dimensional probability distributions across various fields.
- **Role in Reinforcement Learning (RL)**
  - Used in model-based RL to simulate possible futures for planning & decision-making.
  - Enables learning in imaginary environments, reducing risks of real-world errors.
  - Guides exploration by tracking visited states & attempted actions.
  - Supports inverse RL for learning from expert demonstrations.
- **Handling Missing Data & Semi-Supervised Learning**
  - Can train with missing data and predict missing inputs.
  - Enables semi-supervised learning, reducing the need for labeled data.
- **Multi-Modal Learning & Sample Generation**
  - Allows multiple correct outputs for a single input (e.g., video frame prediction).
  - GANs excel in generating realistic samples for various AI applications.

Model Results

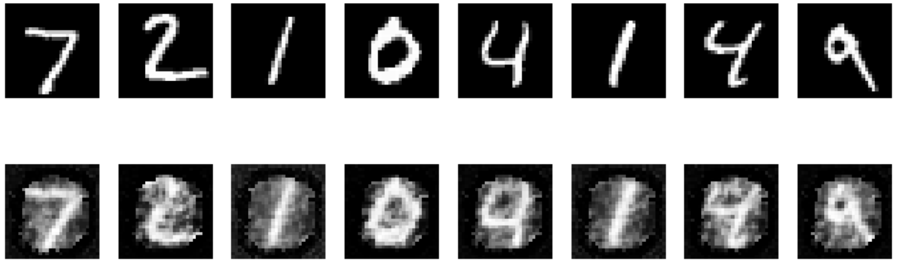
Restricted Boltzmann Machine

Real (Top) vs. RBM Generated (Bottom)



Graph-Structured Network

Real (Top) vs. GSN Generated (Bottom)

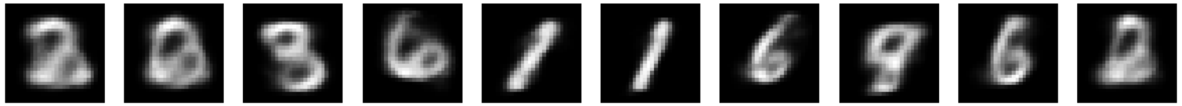


Denoising Autoencoder

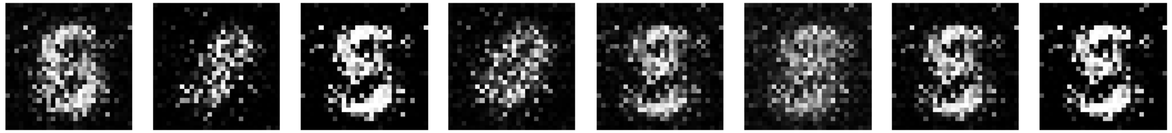
Real (Top) vs. DAE Generated (Bottom)

VAE (generated from noise)

VAE Generated Samples



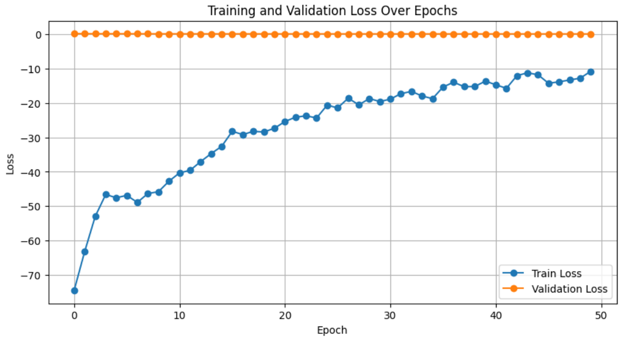
GAN (generated from noise)



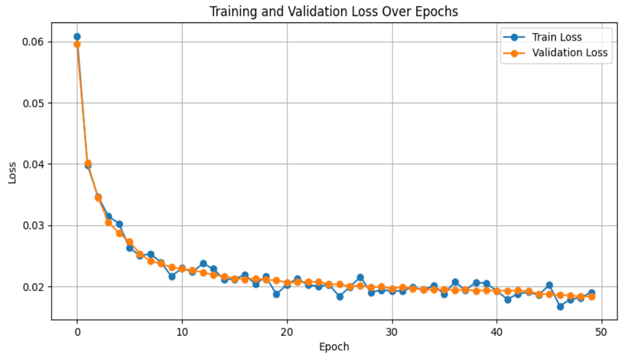
Training & Validation Loss

train_loss = contrastive-divergence free-energy difference

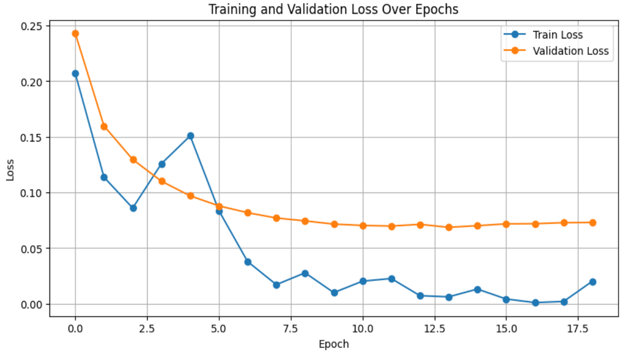
Restricted Boltzmann Machine



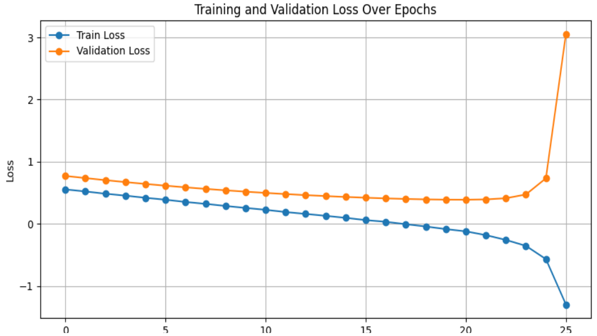
Graph-Structured Network



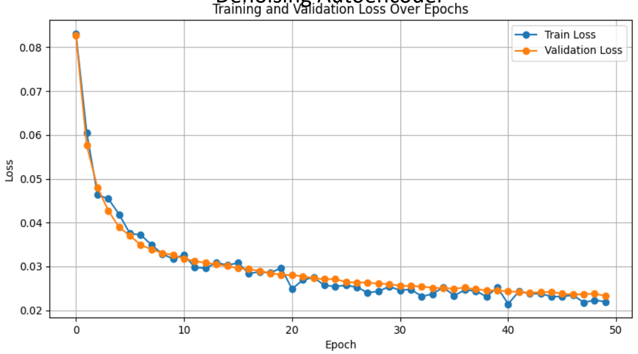
Deep Belief Network (multi-layer RBM - classifier only)



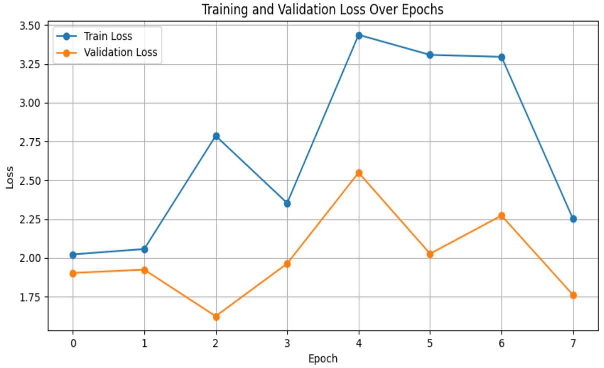
Pixel CNN



Denoising Autoencoder



GAN



Model	**Test_loss**
RBM	0.025985
DBN	0.064641
DAE	0.010321
GSN	0.599898
PIXELCNN	2.370674

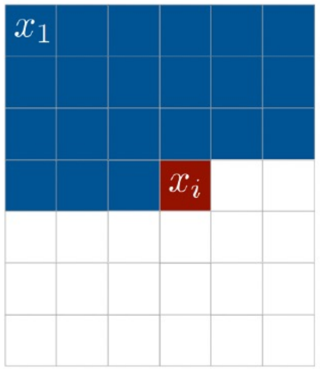
Content

Fully visible belief network (FVBN)

Definition: A Fully Visible Belief Network (FVBN) is a directed probabilistic model where the joint probability of observed variables is factorized using the chain rule of probability.

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

Probability of the i-th pixel value given all previous pixels



Complex distribution over pixel values => Express using a neural network!

Key Features

✅ **Autoregressive structure:** Each variable is modeled sequentially.
✅ **Exact likelihood estimation:** Unlike GANs, FVBNs provide explicit probability distributions.
✅ **No latent variables:** Unlike VAEs or DBNs, FVBNs do not rely on hidden representations.

Applications

📌 **Generative Modeling** – Used in density estimation tasks.
📌 **Sequential Data** – Applied in speech and language models.
📌 **Autoregressive Image Models** – Used in PixelCNN-like architectures.

PixelRNN

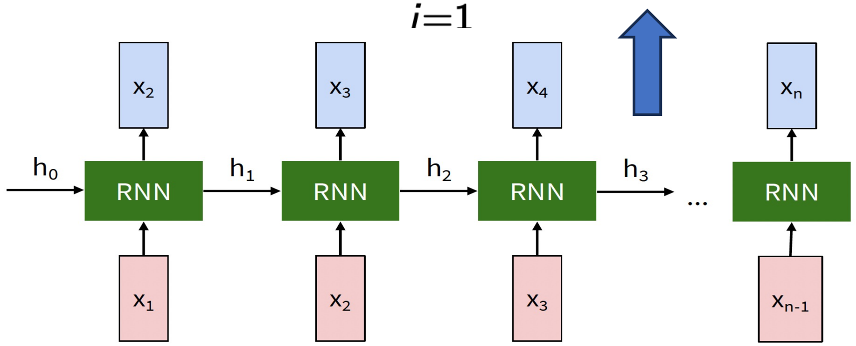
van den Oord et al. (2016).

What is PixelRNN?

- A generative autoregressive model that predicts pixels sequentially along spatial dimensions.
- Used for image generation, inpainting, and density estimation.

Pixel-Level Autoregressive Model

$$P(X) = \prod_{i=1}^{n^2} P(X_i | X_{1:i-1})$$



Color Image

$$P(X_i) = P(X_{i,R}|X_{<i})P(X_{i,G}|X_{<i}, X_{i,R})P(X_{i,B}|X_{<i}, X_{i,R}, X_{i,G})$$

Architecture of PixelRNN

Recurrent Layers (LSTM)

- Row LSTM: Processes the image row-by-row.
- Diagonal BiLSTM: Processes diagonal bands for improved efficiency.

Output Layer

- Uses a softmax layer to predict pixel intensities.

PixelRNN

- Uses LSTMs to process images sequentially.
- Captures long-range dependencies.
- Computationally expensive

Row LSTM

$$h_{i,j} = \text{LSTM}(x_{i,j}, h_{i-1,j})$$

Diagonal BiLSTM

$$h_{i,j} = \text{LSTM}(x_{i,j}, h_{i-1,j-1}, h_{i,j-1}, h_{i-1,j})$$

PixelRNN

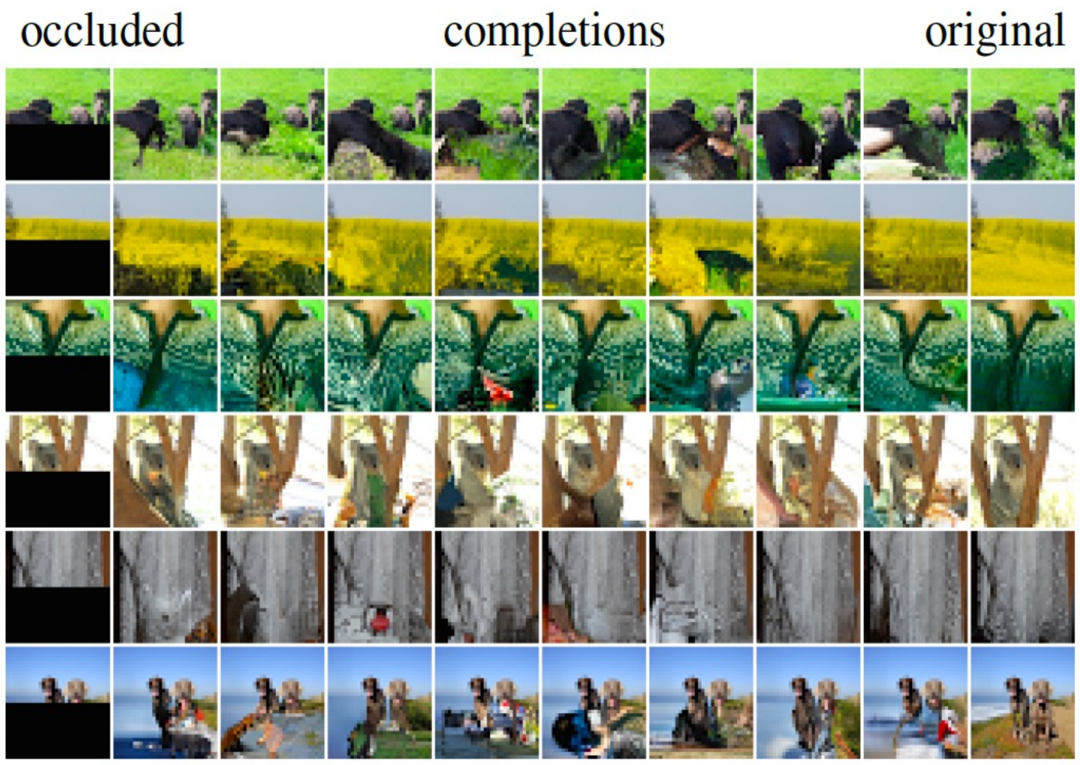
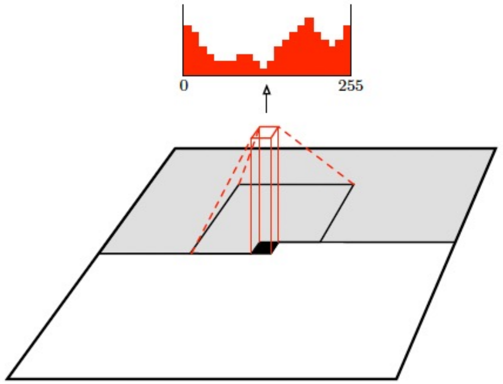


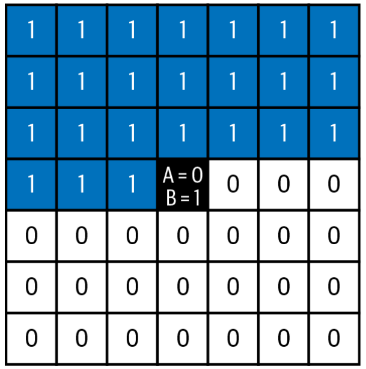
Image completions sampled from a model that was trained on 32x32 ImageNet images (van den Oord et al. (2016)).

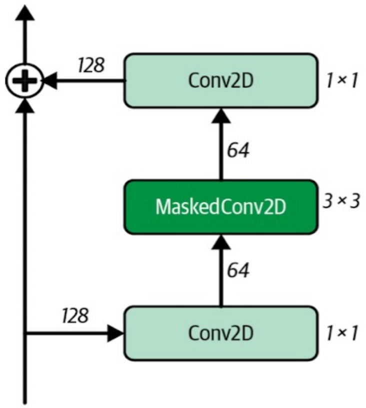
PixelCNN

What is PixelCNN?

- A convolution-based generative model for image synthesis.
- Trains using autoregressive likelihood estimation.
- Faster than PixelRNN due to convolutional structure.
- Efficient for real-world image generation tasks.







- Mask A: Ensures pixels don’t see themselves.
- Mask B: Allows flow of information across layers.

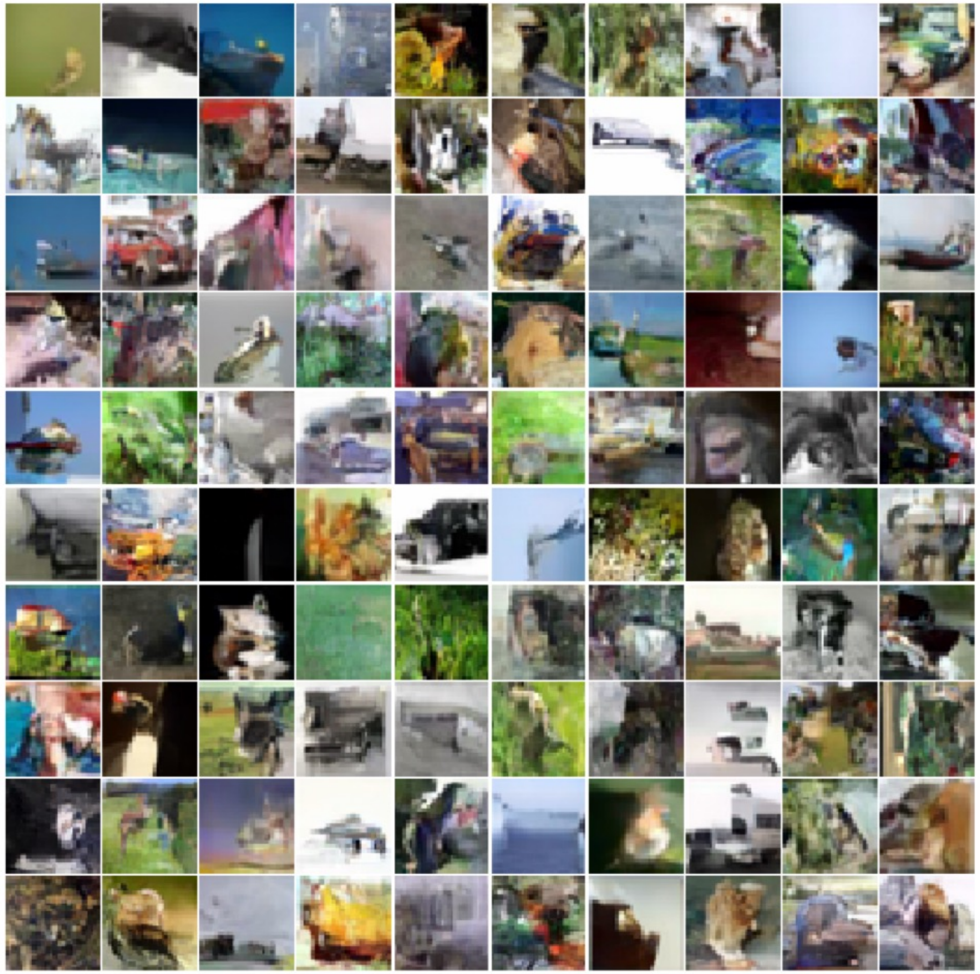
Architecture of PixelCNN

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 pixel_conv_layer (PixelConv  (None, 28, 28, 128)      6400
 Layer)

 residual_block (ResidualBlo  (None, 28, 28, 128)      98624
 ck)

 residual_block_1 (ResidualB  (None, 28, 28, 128)      98624
 lock)

 residual_block_2 (ResidualB  (None, 28, 28, 128)      98624
 lock)

 residual_block_3 (ResidualB  (None, 28, 28, 128)      98624
 lock)

 residual_block_4 (ResidualB  (None, 28, 28, 128)      98624
 lock)

 pixel_conv_layer_6 (PixelCo  (None, 28, 28, 128)      16512
 nvLayer)

 pixel_conv_layer_7 (PixelCo  (None, 28, 28, 128)      16512
 nvLayer)

 conv2d_18 (Conv2D)          (None, 28, 28, 1)         129

=================================================================
Total params: 532,673
Trainable params: 532,673
Non-trainable params: 0
_________________________________________________________________
```

Mask A

Mask B

PixelCNN+++



Model	Bits per sub-pixel
Deep Diffusion (Sohl-Dickstein et al., 2015)	5.40
NICE (Dinh et al., 2014)	4.48
DRAW (Gregor et al., 2015)	4.13
Deep GMMs (van den Oord & Dambre, 2015)	4.00
Conv DRAW (Gregor et al., 2016)	3.58
Real NVP (Dinh et al., 2016)	3.49
PixelCNN (van den Oord et al., 2016b)	3.14
VAE with IAF (Kingma et al., 2016)	3.11
Gated PixelCNN (van den Oord et al., 2016c)	3.03
PixelRNN (van den Oord et al., 2016b)	3.00
PixelCNN++	**2.92**

What are Autoencoders?

Autoencoders are neural networks designed for dimensionality reduction and feature extraction by compressing and reconstructing data.

They consist of two main components:

- **Encoder (e)**: Maps input x to a **low-dimensional latent space** z, where similar inputs have similar latent representations.

  $$e: X \rightarrow Z, \quad z = e(x) \quad \text{with } \dim(X) \gg \dim(Z)$$

- **Decoder (d)**: Reconstructs x from its latent representation z, mapping back to the original input space.

  $$d: Z \rightarrow X \text{ and } \hat{x} = d(z) = d(e(x)).$$

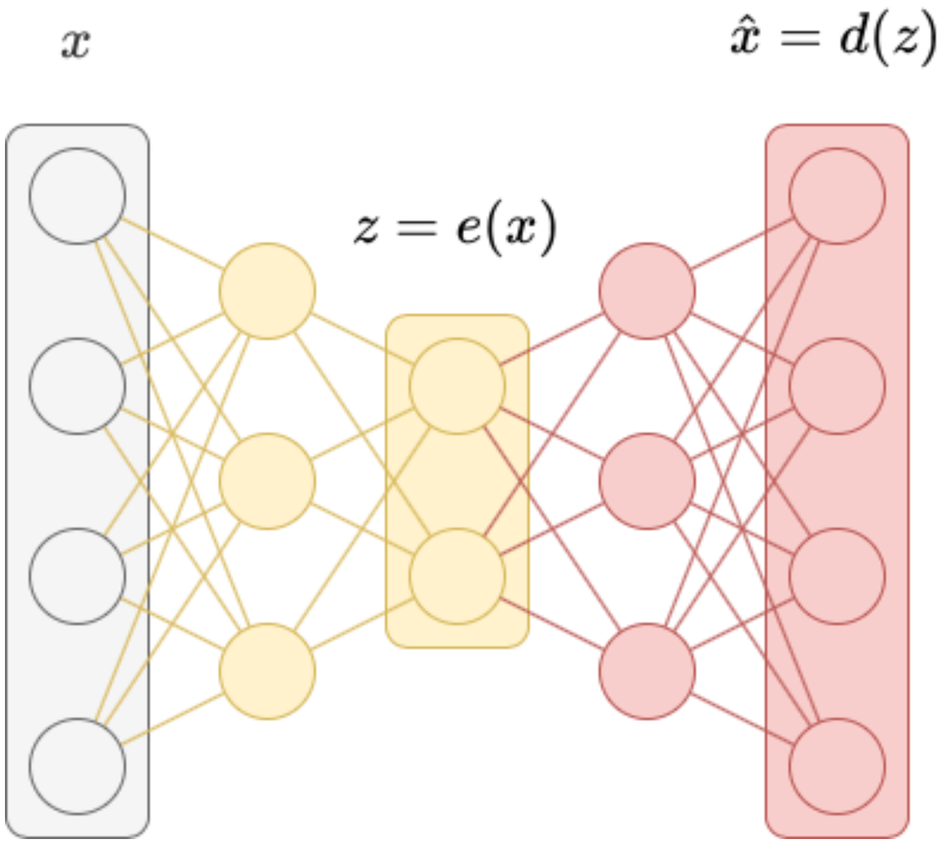
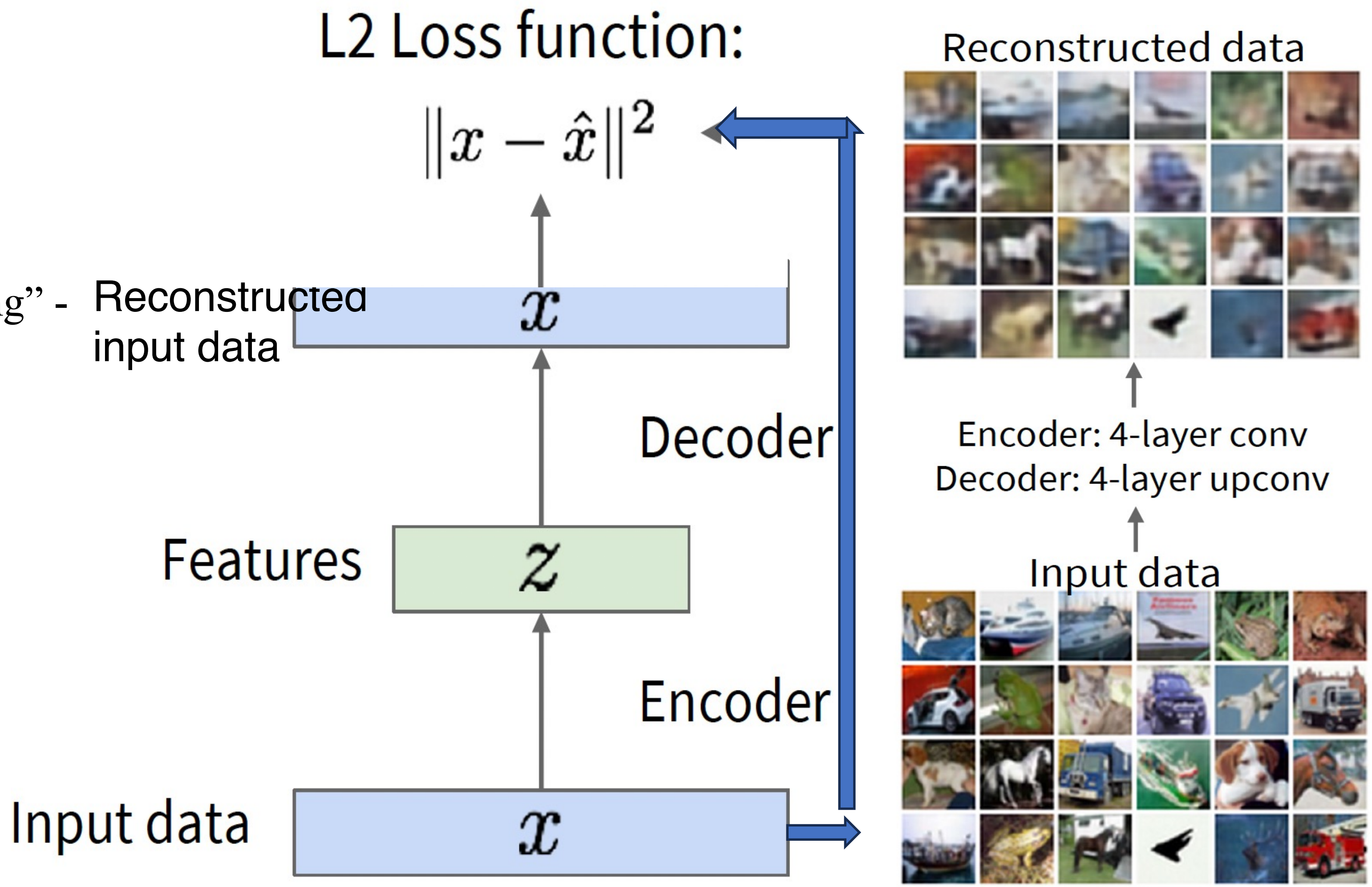


Illustration of autoencoder (source)

What are Autoencoders?

Train such that features can be used to reconstruct original data "Autoencoding" - encoding input itself

Want features to capture meaningful factors of variation in data



Autoencoder Latent Space and Its Limitations

•**Trained on MNIST**, the autoencoder clusters similar digits in the **latent space**.
•**Decoder can reconstruct images** from latent vectors, but gaps in the latent space cause issues.
•**Generative models** aim to produce new samples, but **disjoint latent spaces** in autoencoders make some sampled latent vectors meaningless.
•**Illustration:** In the **top-left corner of the latent space**, unseen regions result in unrealistic reconstructions.
•**Solution: Variational Autoencoders (VAEs)** introduce structured latent spaces to ensure continuity and improve generative performance.
📌 **Key Issue:** Autoencoders are great for representation learning but struggle as generative models due to fragmented latent spaces.

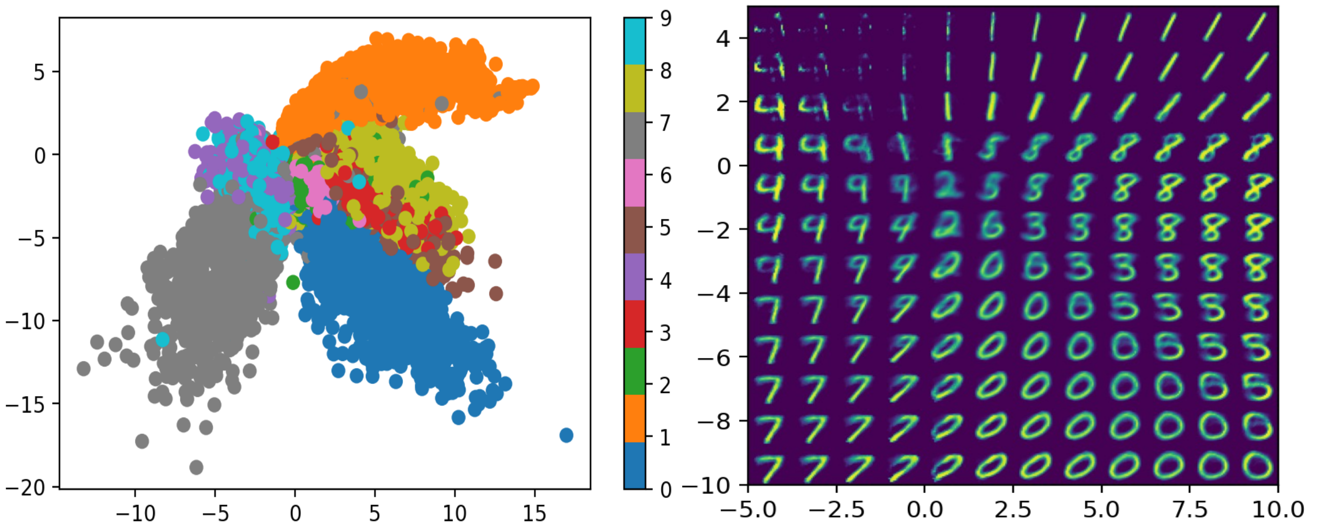


Illustration of example latent vectors using the MNIST dataset (source)

What is a Variational Autoencoder?

📌 **VAE = Autoencoder + Generative Modeling**

- **Same structure as a traditional autoencoder:**
  - ❖ **Encoder:** Compresses input into a latent space representation, but instead of a single point, outputs a probability distribution (Gaussian).
  - ❖ **Decoder:** Samples from this distribution and reconstructs the input.

📌 **Key Difference from Traditional Autoencoders**

❖ Traditional autoencoders map inputs deterministically to a single latent vector z=e(x).

❖ VAEs introduce probabilistic encoding, ensuring smooth and structured latent spaces for better generative performance.

✨ Benefit: Enables meaningful interpolation and sampling for generating new data! 🚀

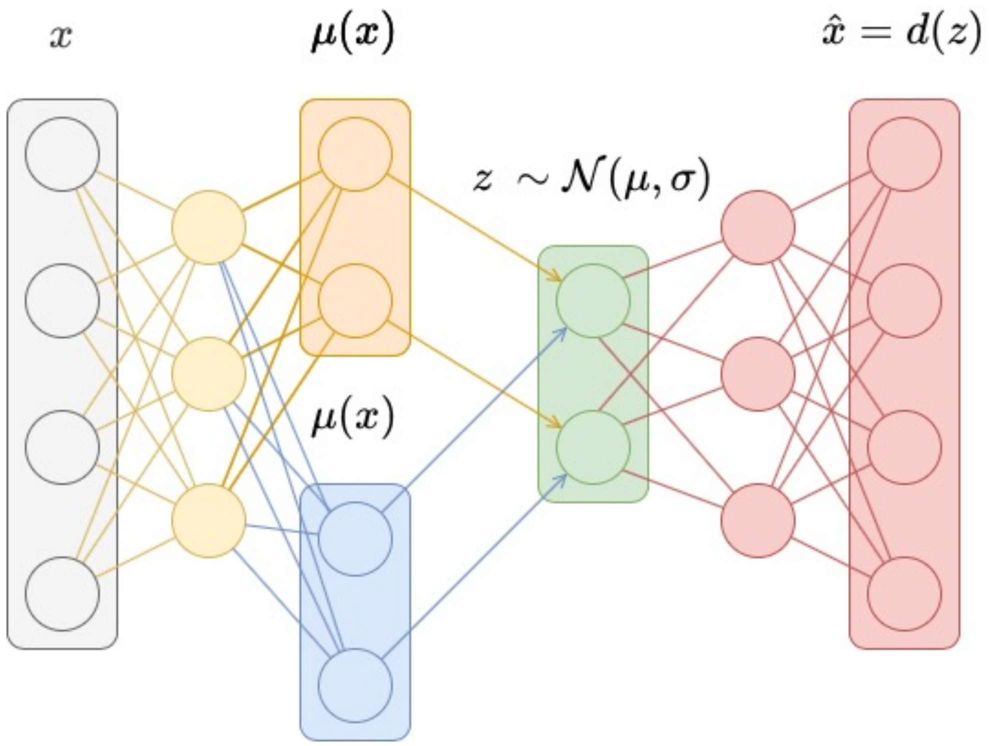


Illustration of VAE (source)

Variational Autoencoder as a DGM

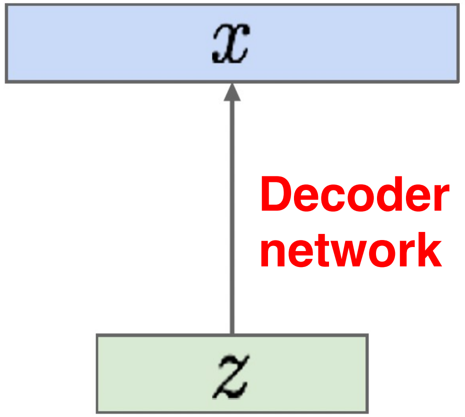
VAEs define an intractable density function with latent

$$p_\theta(x) = \int p_\theta(z) p_\theta(x|z) dz$$

Probabilistic spin on autoencoders - will let us sample from the model to generate data!

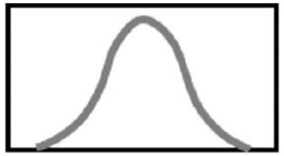
Assume training data $\{x^{(i)}\}_{i=1}^{N}$ is generated from the distribution of unobserved (latent) representation z

Sample from true conditional $p_{\theta^*}(x \mid z^{(i)})$



Conditional p(x|z) is complex (generates image) => represent with neural network

Sample from true prior $z^{(i)} \sim p_{\theta^*}(z)$



Choose prior p(z) to be simple, e.g. Gaussian.

How to train VGE?

Learn model parameters to maximize likelihood of training data

We want to estimate the true parameters of this generative model given training data $\{x^{(i)}\}_{i=1}^{N}$

Q: What is the problem with this?
Intractable!

Data Likelihood

$$p_\theta(x) = \int p_\theta(z) p_\theta(x|z) dz$$

$$\log p(x) \approx \log \frac{1}{k} \sum_{i=1}^{k} p(x|z^{(i)}), \text{ where } z^{(i)} \sim p(z)$$

Intractable to compute p(xlz) for every z!

Monte Carlo estimation is too high variance

Posterior distribution

$$p_\theta(z|x) = p_\theta(x|z) p_\theta(z) / p_\theta(x)$$

Solution: In addition to decoder network modeling $p_\theta(x|z)$, define additional encoder network

$$q_\theta(z|x) \approx p_\theta(z|x)$$

Will see that this allows us to derive a lower bound on the data likelihood that is tractable, which we can optimize.

How to approximate VGE?

$$
\begin{aligned}
\log p_\theta(x^{(i)}) &= \mathbf{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta(x^{(i)})\right] \quad (p_\theta(x^{(i)}) \text{ Does not depend on } z) \\
&= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\right] \quad (\text{Bayes' Rule}) \\
&= \mathbf{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})} \frac{q_\phi(z \mid x^{(i)})}{q_\phi(z \mid x^{(i)})}\right] \quad (\text{Multiply by constant}) \\
&= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z)}\right] + \mathbf{E}_z\left[\log \frac{q_\phi(z \mid x^{(i)})}{p_\theta(z \mid x^{(i)})}\right] \quad (\text{Logarithms}) \\
&= \mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z)) + D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z \mid x^{(i)}))
\end{aligned}
$$

Decoder network gives pθ(x|z), can compute estimate of this term through sampling (need some trick to differentiate through sampling).

This KL term (between Gaussians for encoder and z prior) has nice closed-form solution!

pθ(z|x) intractable (saw earlier), can't compute this KL term. But we know KL divergence always >= 0.

How to approximate VGE?

Decoder: reconstruct the input data

Encoder: make approximate posterior distribution. close to prior

$$\log p_\theta(x^{(i)}) = \boxed{\underbrace{\mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z))}_{\mathcal{L}(x^{(i)},\theta,\phi)}} + \underbrace{D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z \mid x^{(i)}))}_{\boxed{\geq 0}}$$

We want to maximize the data likelihood.

Tractable lower bound which we can take gradient of and optimize! (pθ(x|z) differentiable, KL term differentiable)

Variational evidence lower bound (ELBO):

$$\mathcal{L}(x,\theta,\phi) \leq \log p_\theta(x)$$

Training: Maximize lower bound

$$\hat{\theta},\hat{\phi} = \arg\max_{\theta,\phi} \sum_{i=1}^{n} \mathcal{L}(x_i,\theta,\phi)$$

Stochastic Optimization of ELBO

Algorithm 1: Stochastic optimization of the ELBO. Since noise originates from both the minibatch sampling and sampling of $p(\boldsymbol{\epsilon})$, this is a doubly stochastic optimization procedure. We also refer to this procedure as the *Auto-Encoding Variational Bayes* (AEVB) algorithm.

Data:
- $\mathcal{D}$: Dataset
- $q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})$: Inference model
- $p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z})$: Generative model

Result:
- $\boldsymbol{\theta}, \boldsymbol{\phi}$: Learned parameters

$(\boldsymbol{\theta}, \boldsymbol{\phi}) \leftarrow$ Initialize parameters
while *SGD not converged* **do**
- $\mathcal{M} \sim \mathcal{D}$ (Random minibatch of data)
- $\boldsymbol{\epsilon} \sim p(\boldsymbol{\epsilon})$ (Random noise for every datapoint in $\mathcal{M}$)
- Compute $\tilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M}, \boldsymbol{\epsilon})$ and its gradients $\nabla_{\boldsymbol{\theta},\boldsymbol{\phi}}\tilde{\mathcal{L}}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathcal{M}, \boldsymbol{\epsilon})$
- Update $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ using SGD optimizer

end

$$\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right]$$

$$\nabla_{\boldsymbol{\theta}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \nabla_{\boldsymbol{\theta}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right]$$

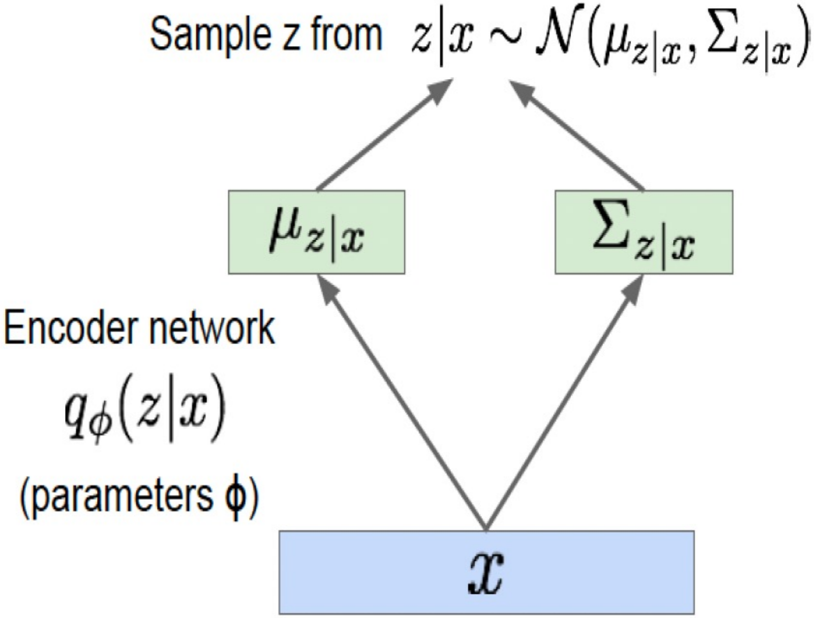
$$\nabla_{\boldsymbol{\phi}}\mathcal{L}_{\boldsymbol{\theta},\boldsymbol{\phi}}(\mathbf{x}) = \nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[\log p_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) - \log q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})\right]$$

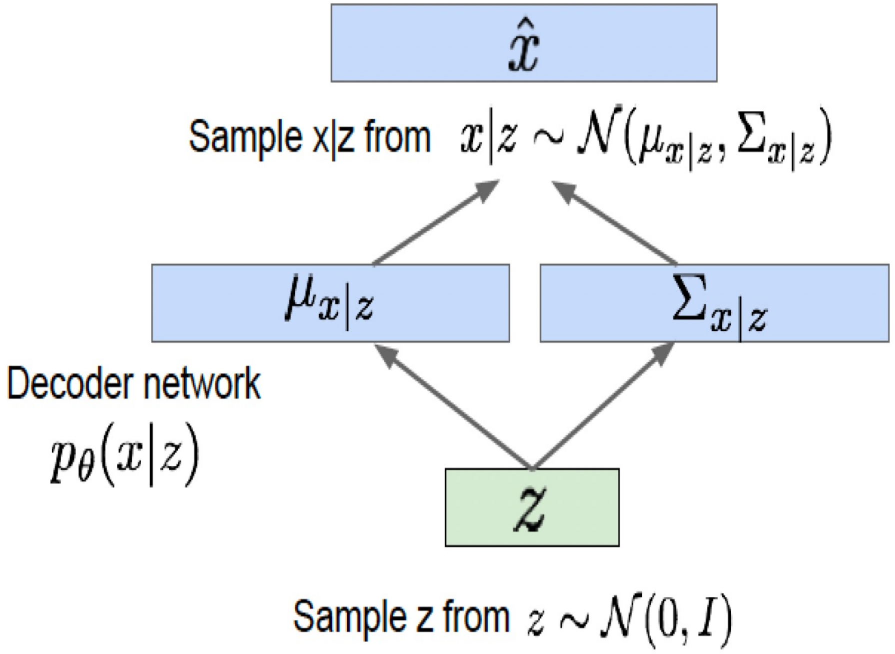
Reparametrization Trick: $\mathbf{z} = \mathbf{g}(\boldsymbol{\epsilon}, \boldsymbol{\phi}, \mathbf{x})$

$$\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[f(\mathbf{z})\right] = \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[f(\mathbf{z})\right]$$

$$\begin{aligned}\nabla_{\boldsymbol{\phi}}\mathbb{E}_{q_{\boldsymbol{\phi}}(\mathbf{z}|\mathbf{x})}\left[f(\mathbf{z})\right] &= \nabla_{\boldsymbol{\phi}}\mathbb{E}_{p(\boldsymbol{\epsilon})}\left[f(\mathbf{z})\right] \\ &= \mathbb{E}_{p(\boldsymbol{\epsilon})}\left[\nabla_{\boldsymbol{\phi}}f(\mathbf{z})\right] \\ &\simeq \nabla_{\boldsymbol{\phi}}f(\mathbf{z})\end{aligned}$$

A Theoretical Example

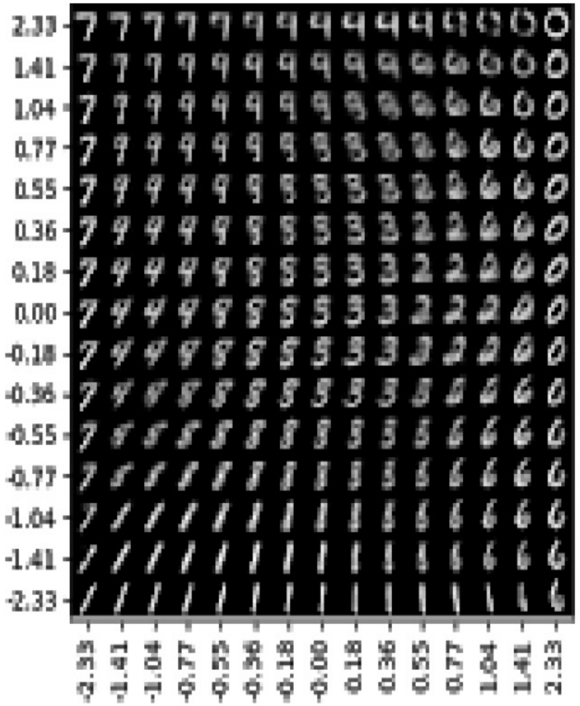


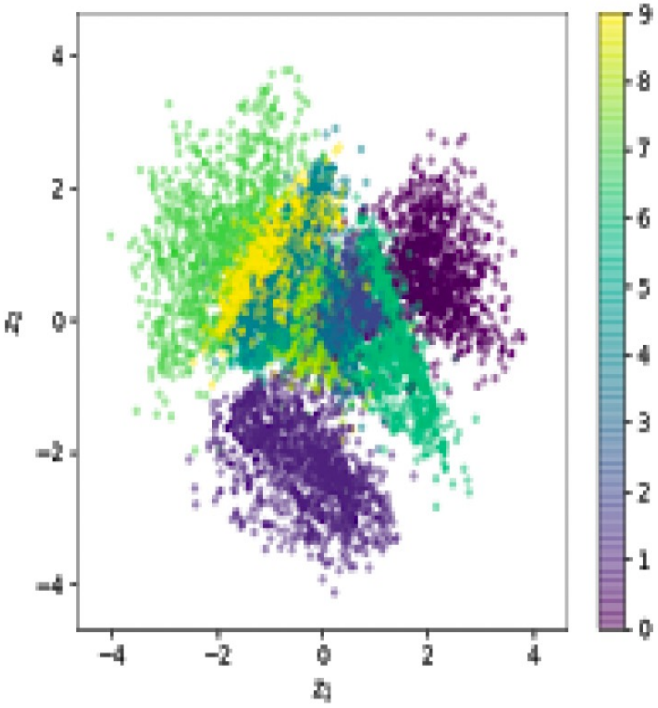


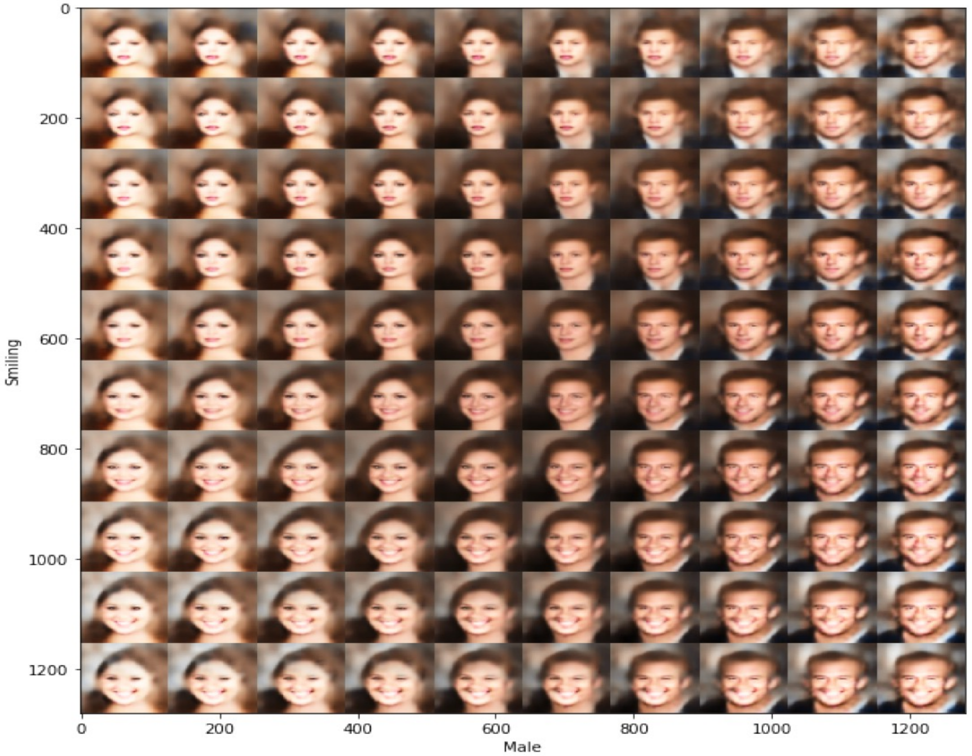
$$\int q_{\boldsymbol{\theta}}(\mathbf{z}) \log p(\mathbf{z})\, d\mathbf{z} = \int \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\sigma}^2) \log \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})\, d\mathbf{z}$$
$$= -\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}(\mu_j^2 + \sigma_j^2)$$

$$\begin{aligned} \log p(\mathbf{x}|\mathbf{z}) &= \log \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I}) \\ \text{where } \boldsymbol{\mu} &= \mathbf{W}_4\mathbf{h} + \mathbf{b}_4 \\ \log \boldsymbol{\sigma}^2 &= \mathbf{W}_5\mathbf{h} + \mathbf{b}_5 \\ \mathbf{h} &= \tanh(\mathbf{W}_3\mathbf{z} + \mathbf{b}_3) \end{aligned}$$

Real Examples







Code: Vanilla VAE

```
# ------ Define the Encoder & Decoder ------
# The encoder outputs two vectors: mu (μ) — the mean of the latent distribution & logvar (logσ²) — the log-variance of the latent distribution. Reparameterization Trick is used: z = μ + exp(1/2 logσ²)⊙ϵ, ϵ ~ N(0,1)
# The decoder maps the latent vector z back to the original data space. We will use a final Sigmoid to produce pixel intensities in [0,1].
class VAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim):
        super(VAE, self).__init__()
        # Encoder
        self.enc_fc1 = nn.Linear(input_dim, hidden_dim)
        self.enc_fc2_mu = nn.Linear(hidden_dim, latent_dim)
        self.enc_fc2_logvar = nn.Linear(hidden_dim, latent_dim)
        # Decoder
        self.dec_fc1 = nn.Linear(latent_dim, hidden_dim)
        self.dec_fc2 = nn.Linear(hidden_dim, input_dim)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()
    def encoder(self, x):
        """Encode the input into latent parameters (mu, logvar)."""
        h = self.relu(self.enc_fc1(x))
        mu = self.enc_fc2_mu(h)
        logvar = self.enc_fc2_logvar(h)
        return mu, logvar
    def reparameterize(self, mu, logvar):
        """Reparameterization trick to sample z."""
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)   # same shape as std
        z = mu + eps * std
        return z
```

Code: Vanilla VAE

```
    def decoder(self, z):
        """Decode the latent vector into reconstructed input."""
        h = self.relu(self.dec_fc1(z))
        x_recon = self.sigmoid(self.dec_fc2(h))
        return x_recon

    def forward(self, x):
        """Forward pass: encoder -> reparam -> decoder."""
        mu, logvar = self.encoder(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decoder(z)
        return x_recon, mu, logvar

# ----- Define the loss function -----
# The total loss is the sum of Reconstruction Loss: Typically we use Binary Cross Entropy between the reconstructed
image and the original image (scaled to [0,1]) and KL Divergence Loss: Encourages the approximate posterior qϕ(z|x) to
be close to the prior p(z)=N(0,I).
def loss_function(x_recon, x, mu, logvar):
    # Reconstruction loss (assuming x, x_recon in [0, 1])
    bce = nn.functional.binary_cross_entropy(
        x_recon, x, reduction='sum'
    )  # sum over all pixels
    # KL Divergence
    # KL(N(mu, sigma^2) || N(0,1))
    # = 0.5 * sum(exp(logvar) + mu^2 - 1 - logvar)
    kl = 0.5 * torch.sum(torch.exp(logvar) + mu**2 - 1.0 - logvar)
    return bce + kl
```

Strengths & Limitations

📌 **Key Idea:**

➢ Adds a probabilistic spin to traditional autoencoders, enabling data generation.

➢ Defines an intractable density, requiring variational inference to derive and optimize a lower bound (ELBO).

📌 **Pros:**

✔ Principled generative approach based on probabilistic modeling.

✔ Interpretable latent space enables meaningful structure in representations.

✔ Inference of q(z|x) allows feature extraction for other tasks.

📌 **Cons:**

❌ Optimizes a lower bound on likelihood, which may not be an ideal evaluation metric.

❌ Lower sample quality compared to PixelRNN/PixelCNN.

❌ Blurry reconstructions compared to GANs, which generate sharper images.

📌 **Active Research Areas:**

🔷 Flexible Approximate Posteriors: Moving beyond diagonal Gaussian assumptions to richer models like Gaussian Mixture Models (GMMs) or Categorical Distributions.

🔷 Disentangled Representations: Learning independent latent factors for better interpretability.

🔷 Improving Training Objectives: Hybrid models incorporating adversarial learning (VAE-GANs).

🚀 **Future Directions**: Enhancing sample quality while retaining VAE's structured latent space!

Content

“This (GANS), and the variations that are now being proposed is the most interesting idea in the last 10 years in ML, in my opinion”

–Yann LeCun

What is GAN?

Problem:

- ❖ We want to sample from a complex, high-dimensional training distribution.
- ❖ There is no direct way to explicitly learn or model the data distribution.

Solution:

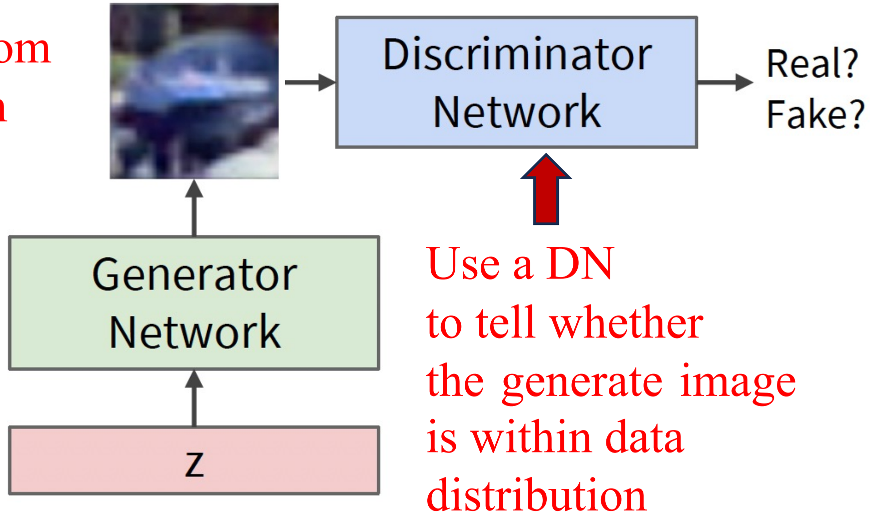
- ➢ Instead of learning the distribution explicitly, GANs learn a transformation.
- ➢ Start by sampling from a simple distribution (e.g., Gaussian noise).
- ➢ Train a neural network to transform the simple distribution into the training data distribution.

Key Idea:

- ✓ GANs learn to generate new samples indirectly through adversarial training.
- ✓ The model never explicitly estimates the probability density function of the data.

Objective: generated images should look "real"

Output: Sample from training distribution



Use a DN to tell whether the generate image is within data distribution ("real") or not

Input: Random noise

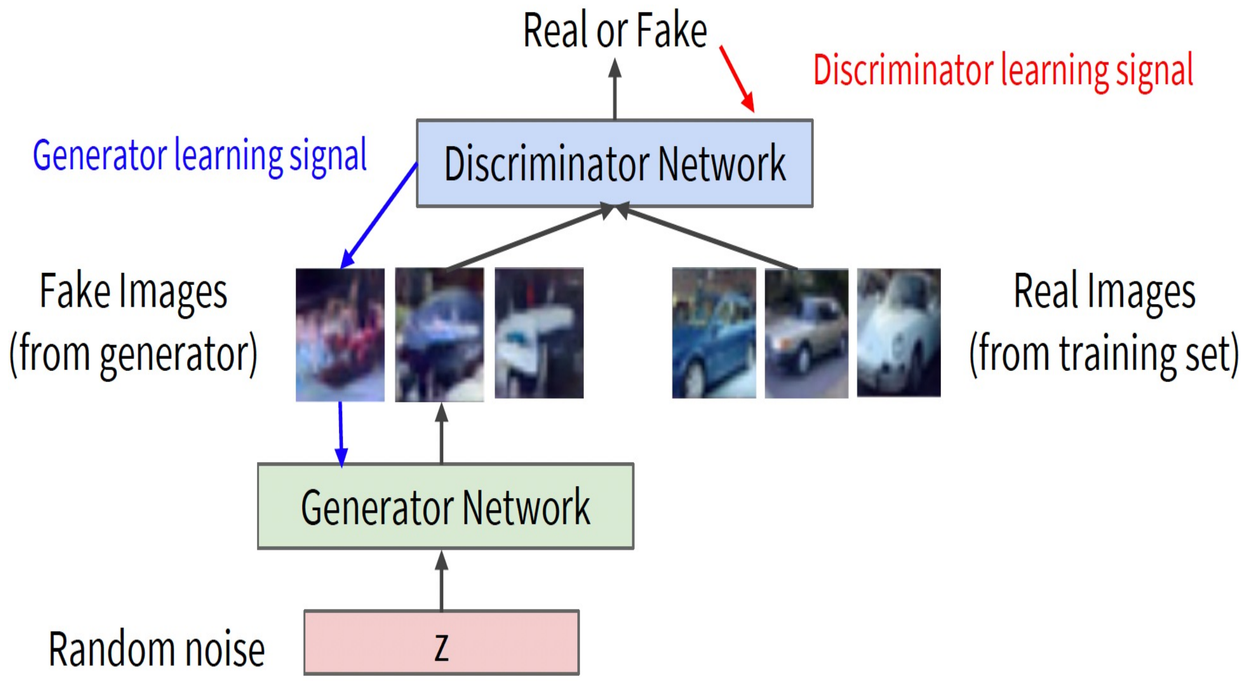
Generative Vs Discriminative Models

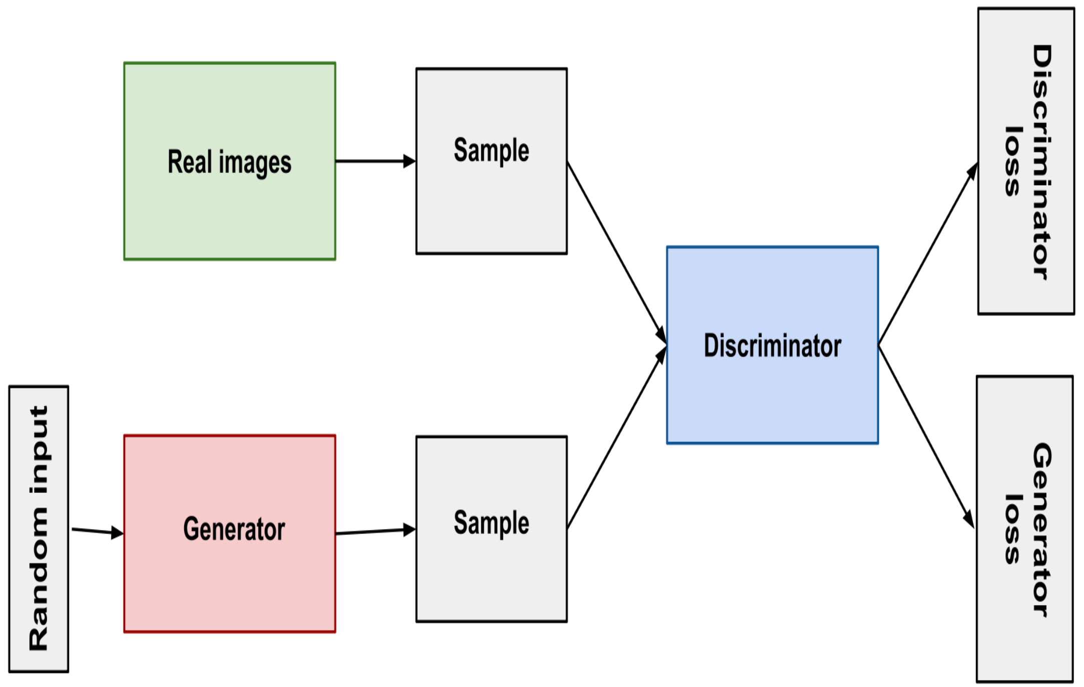
Generative Models:

- Can generate new data samples resembling real data.
- Example: GANs generate realistic images that resemble real ones.

Discriminative Models:

- Focus on classification by distinguishing between different categories.
- Example: A decision tree can classify dogs and cats but cannot generate them.





Overview of GAN Training

Discriminator Network: Tries to distinguish between real and fake images.
Generator Network: Tries to fool the discriminator by generating real-looking images.
Training Process: Both networks are trained jointly in a minimax game.

Minimax objective function:

$$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \underbrace{D_{\theta_d}(x)}_{\text{Discriminator output for real data x}} + \mathbb{E}_{z \sim p(z)} \log(1 - \underbrace{D_{\theta_d}(G_{\theta_g}(z))}_{\text{Discriminator output for generated fake data G(z)}}) \right]$$

Generator objective

Discriminator objective

- **Discriminator** (θ_d) wants to maximize the objective such that $D(x) \approx 1$ (real) and $D(G(z)) \approx 0$ (fake).
- **Generator** (θ_g) wants to minimize the objective such that $D(G(z)) \approx 1$ (fooling the discriminator).

GAN Training

Alternate between:

1. Gradient ascent on discriminator

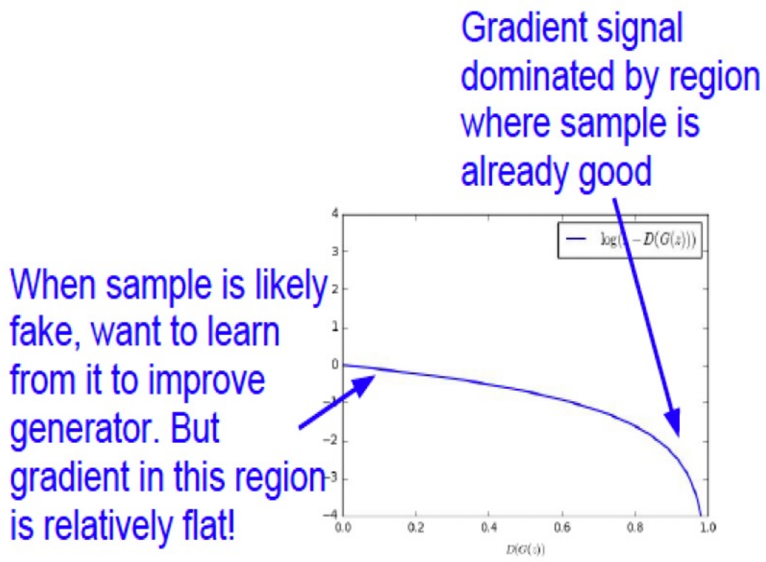
$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log D_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right]$$

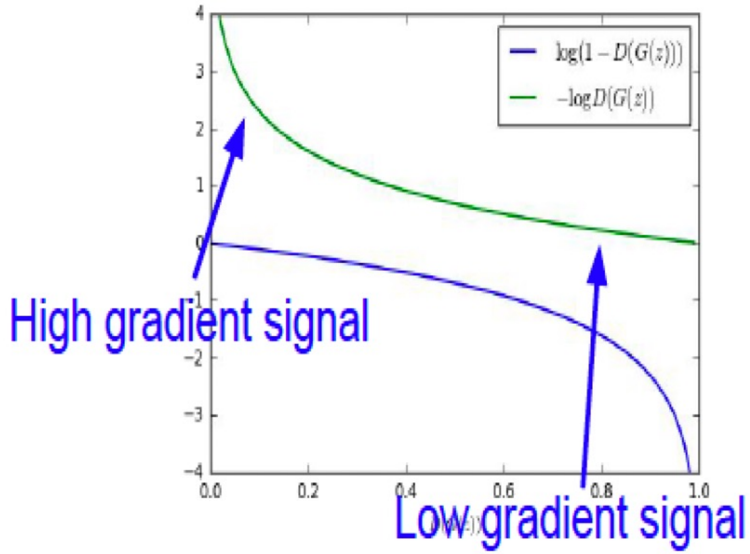
2. Gradient descent on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D_{\theta_d}(G_{\theta_g}(z)))$$

Instead: Consider a different objective

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(D_{\theta_d}(G_{\theta_g}(z)))$$





GAN Algorithms

for number of training iterations **do**

for k steps **do**

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

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)}))) \right]$$

end for

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

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log(D_{\theta_d}(G_{\theta_g}(z^{(i)})))$$

end for

Alternating Training for GANs

GAN Training Process (Alternating Phases):

1. Discriminator Training: Trains for one or more epochs while the generator remains unchanged. It learns to differentiate real from generated data, adapting to the generator's flaws.

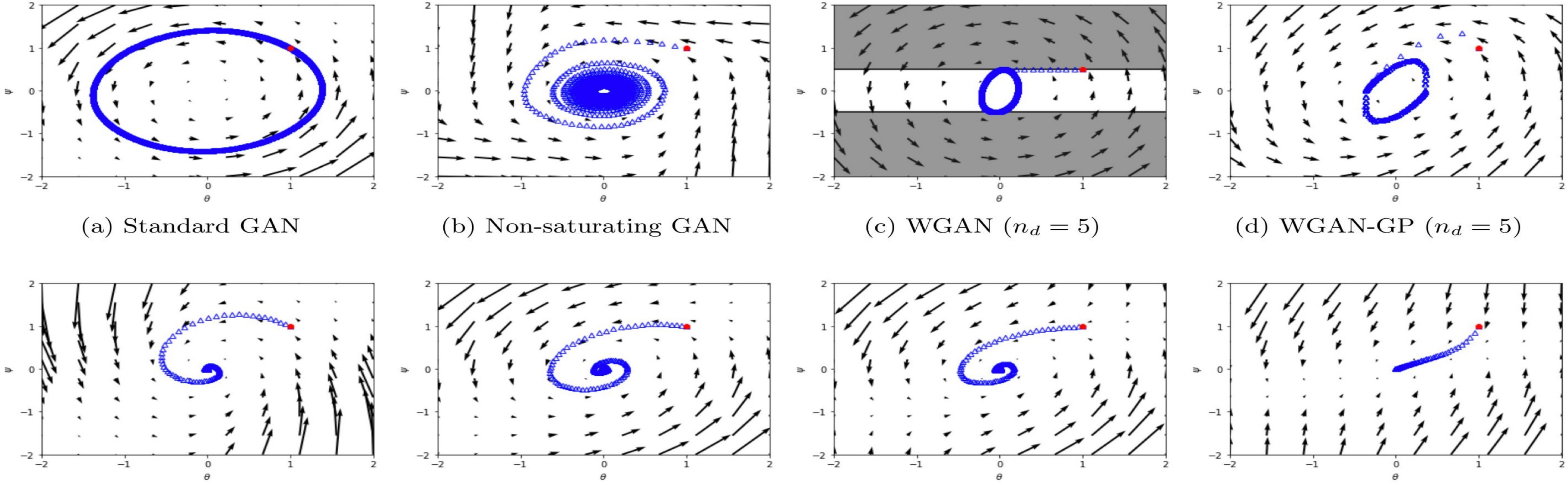
2. Generator Training: Trains for one or more epochs while the discriminator remains unchanged. This prevents the generator from chasing a moving target.

Training Dynamics:

- As the generator improves, the discriminator struggles to distinguish real from fake data.
- A perfect generator results in a discriminator with 50% accuracy (random guessing).
- Overtraining can degrade performance, leading to unstable convergence where the generator receives meaningless feedback.

Challenges in GAN Training

- **Hyperparameter Sensitivity:** GANs are sensitive to learning rates, batch sizes, and architectural choices.
- **Mode Collapse:** The generator produces limited diversity.
- **Training Instability:** The minimax optimization is difficult to balance.
- **Vanishing/Exploding Gradients:** The discriminator can become too strong or weak, leading to poor gradients.
- **Non-Convergence:** The model oscillates instead of converging.



(a) Standard GAN (b) Non-saturating GAN (c) WGAN ($n_d = 5$) (d) WGAN-GP ($n_d = 5$)

(e) Consensus optimization (f) Instance noise (g) Gradient penalty (h) Gradient penalty (CR)

Hyperparameter Sensitivity

1. Adjust Learning Rates Carefully

- **Learning Rate (α)**: Too high → instability, Too low → slow convergence.
- **Two-Timescale Update Rule (TTUR)**: Use a smaller learning rate for the generator than the discriminator to balance training.

2. Tune Adam Hyperparameters

- Standard settings **(β1=0.9, β2=0.999)** can lead to oscillations.
- For **GANs**, reducing β1 to **0.5** improves stability.

3. Normalize Inputs and Use Spectral Normalization

- ✓ Normalize training images between **[−1,1]** instead of [0,1] (for Tanh activation).
- ✓ Use **Spectral Normalization** on the discriminator to control weight magnitudes.

4. Improve Loss Functions

- **Wasserstein Loss (WGAN)**: Uses Earth-Mover distance for better gradient behavior.
- **Gradient Penalty (WGAN-GP)**: Adds stability and prevents exploding gradients:

5. Use Progressive Training

- **Start with low-resolution images**, gradually increasing resolution (used in **Progressive Growing GANs**).
- Helps GAN **learn simple features first** before complex details.

Hyperparameter Sensitivity

6. Apply Regularization Techniques

- **Batch Normalization**: Helps control variance, but can cause mode collapse in GANs.
- **Instance Normalization**: Often more stable than batch normalization.
- **Dropout in Discriminator**: Helps prevent overfitting.

7. Monitor Convergence and Use Early Stopping

- **Track GAN metrics** (FID, Inception Score) instead of just loss values.
- **Avoid overtraining**: If the discriminator gets too strong, **freeze it temporarily**.

8. Use Larger Batch Sizes

- GANs often benefit from **larger batch sizes** (e.g., 128–512) to stabilize updates.
- **Gradient accumulation** can be used if GPU memory is limited.

9. Data Augmentation

- **Apply transformations (rotation, flipping, color jitter)** to make training more robust.
- Prevents the discriminator from memorizing training data.

10. Experiment with Alternative Architectures

- **Self-Attention GANs (SAGAN)**: Improves global structure modeling.
- **BigGAN**: Uses **larger batch sizes** and **orthogonal regularization** for stability.

Mode Collapse

Mode Collapse in GANs refers to a common failure mode where the **generator** fails to capture the full diversity of the data distribution and produces **limited variations** of samples. Instead of generating a wide range of outputs, it collapses to generating a few or even a single type of sample repeatedly.

Why Does Mode Collapse Occur?

- ❖ **Imbalanced Generator-Discriminator Learning**
- ❖ **Training Instability**
- ❖ **Lack of Diversity-Promoting Mechanisms**

Effects of Mode Collapse

- ❖ **Reduced Sample Diversity** → Poor representation of the real dataset.
- ❖ **Low-Quality Generation** → Outputs look repetitive and lack variety.
- ❖ **Unreliable Model** → The generator fails to generalize.

Illustration of example monotonous output. (source)

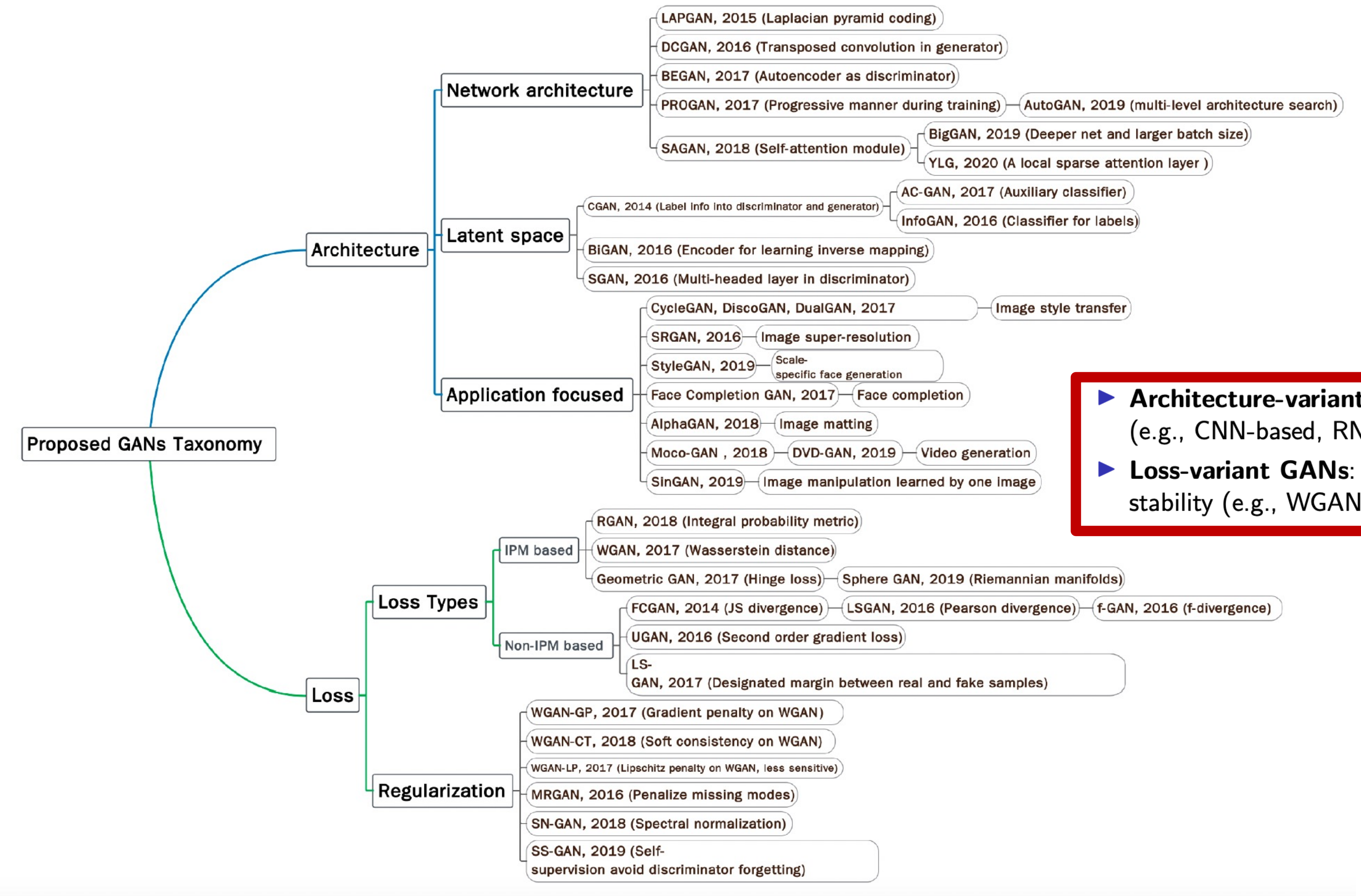
Techniques to Mitigate Mode Collapse

1. **Minibatch Discrimination**
   Encourages diversity by comparing samples in each batch.
2. **Feature Matching**
   Instead of just fooling the discriminator, the generator learns to match feature statistics of real data.
3. **Wasserstein GAN (WGAN)**
   Uses the Earth Mover (Wasserstein) distance to **stabilize training** and avoid collapsing to few modes.
4. **Unrolled GANs**
   Allows the generator to anticipate discriminator updates, preventing it from getting stuck in mode collapse.
5. **Mutual Information Regularization**
   Forcing the generator to learn **meaningful latent representations** that generate diverse outputs.

.

More details of example GAN suffering mode collapse: https://neptune.ai/blog/gan-failure-modes

The taxonomy of the recent GANs

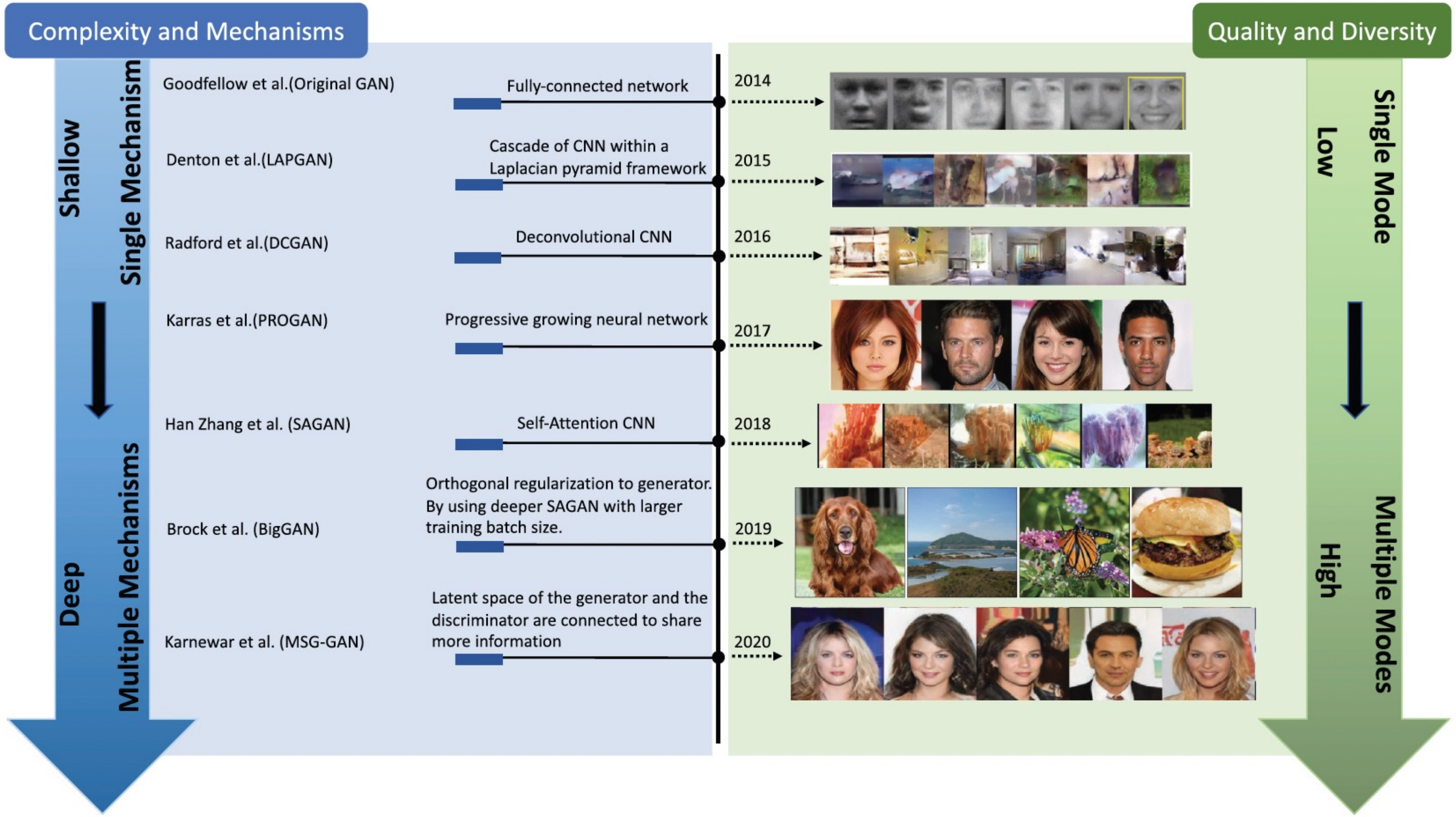


- **Architecture-variant GANs**: Modify network structures (e.g., CNN-based, RNN-based models).
- **Loss-variant GANs**: Modify loss functions to improve stability (e.g., WGAN, LSGAN, f-GAN).

Different GANs

Model	Stability	Mode Collapse	Convergence	Sample Quality	Special Features
GAN	Low	High	Unstable	Medium	Baseline
LSGAN	Medium	Medium	More stable	Medium	Least squares loss
WGAN	High	Low	More stable	High	Wasserstein distance
WGAN-GP	Very High	Very Low	Very stable	Very High	Gradient Penalty
cGAN	Medium	Medium	Stable	High	Class conditioning
StyleGAN	High	Low	Stable	Very High	Style control

Timeline of GAN architectures



Complexity in blue stream refers to size of the architecture and computational cost such as batch size. Mechanisms refer to the number of types of operations (e.g., convolution, deconvolution, self-attention) used in the architecture (e.g., FCGAN uses fully connected layers for both discriminator and generator. In this case, the value for mechanisms is 1).

Loss-variant GANs

- Jointly training two networks is challenging, can be unstable. Choosing objectives with better loss landscapes helps training, and is an active area of research.
- $X \sim P_X$ vs. $G(Z) \sim P_G$ with $Z \sim N(0, I)$.
- Training GAN is equivalent to minimizing Jensen-Shannon divergence between generator and data distributions.
- $D(P_X, P_G) = \sup_{f \in \mathcal{F}} \left\{ \mathbb{E}_{X \sim P_X} \phi_1(f(X)) - \mathbb{E}_{Y \sim P_G} \phi_2(f(Y)) \right\}$

GAN	Discriminator Loss	Generator Loss
MM GAN	$\mathcal{L}_D^{GAN} = -\mathbb{E}_{x \sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$	$\mathcal{L}_G^{GAN} = \mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$
NS GAN	$\mathcal{L}_D^{NSGAN} = -\mathbb{E}_{x \sim p_d}[\log(D(x))] - \mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$	$\mathcal{L}_G^{NSGAN} = -\mathbb{E}_{\hat{x} \sim p_g}[\log(D(\hat{x}))]$
WGAN	$\mathcal{L}_D^{WGAN} = -\mathbb{E}_{x \sim p_d}[D(x)] + \mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})]$	$\mathcal{L}_G^{WGAN} = -\mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})]$
WGAN GP	$\mathcal{L}_D^{WGANGP} = \mathcal{L}_D^{WGAN} + \lambda \mathbb{E}_{\hat{x} \sim p_g}[(\|\|\nabla D(\alpha x + (1 - \alpha \hat{x})\|\|_2 - 1)^2]$	$\mathcal{L}_G^{WGANGP} = -\mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})]$
LS GAN	$\mathcal{L}_D^{LSGAN} = -\mathbb{E}_{x \sim p_d}[(D(x) - 1)^2] + \mathbb{E}_{\hat{x} \sim p_g}[D(\hat{x})^2]$	$\mathcal{L}_G^{LSGAN} = -\mathbb{E}_{\hat{x} \sim p_g}[(D(\hat{x} - 1))^2]$
DRAGAN	$\mathcal{L}_D^{DRAGAN} = \mathcal{L}_D^{GAN} + \lambda \mathbb{E}_{\hat{x} \sim p_d + \mathcal{N}(0,c)}[(\|\|\nabla D(\hat{x})\|\|_2 - 1)^2]$	$\mathcal{L}_G^{DRAGAN} = \mathbb{E}_{\hat{x} \sim p_g}[\log(1 - D(\hat{x}))]$
BEGAN	$\mathcal{L}_D^{BEGAN} = \mathbb{E}_{x \sim p_d}[\|\|x - AE(x)\|\|_1] - k_t \mathbb{E}_{\hat{x} \sim p_g}[\|\|\hat{x} - AE(\hat{x})\|\|_1]$	$\mathcal{L}_G^{BEGAN} = \mathbb{E}_{\hat{x} \sim p_g}[\|\|\hat{x} - AE(\hat{x})\|\|_1]$

GAN-related Loss Functions

GANs aim to approximate the real data distribution $p_{data}(x)$ using a generator network $G(\eta)$, where $\eta \sim p_z(\eta)$ is drawn from a simple prior distribution.

- The training process is driven by a discriminator D(x), which distinguishes real from generated samples.
- The loss function should measure **the divergence** between $p_{data}(x)$ and $p_g(x)$, where $p_g(x)$ is the distribution of generated samples

Minimax Loss: In the original GANs, the generator tries to minimize the following function while the discriminator tries to maximize it:

$$\min_G \max_D V(D,G) = E_x[\log(D(x))] + E_z[\log(1 - D(G(z)))]$$

The formula derives from the cross-entropy between the real and generated distributions.

$$\max_D V(D,G) = \max_D \{\mathbb{E}_{x\sim P_{\text{data}}}[\log D(x)] + \mathbb{E}_{x\sim P_g}[\log(1 - D(x))]\}$$

For a given x, the optimal discriminator is given by

$$D^*(x) = \frac{P_{\text{data}}(x)}{P_{\text{data}}(x) + P_g(x)}$$

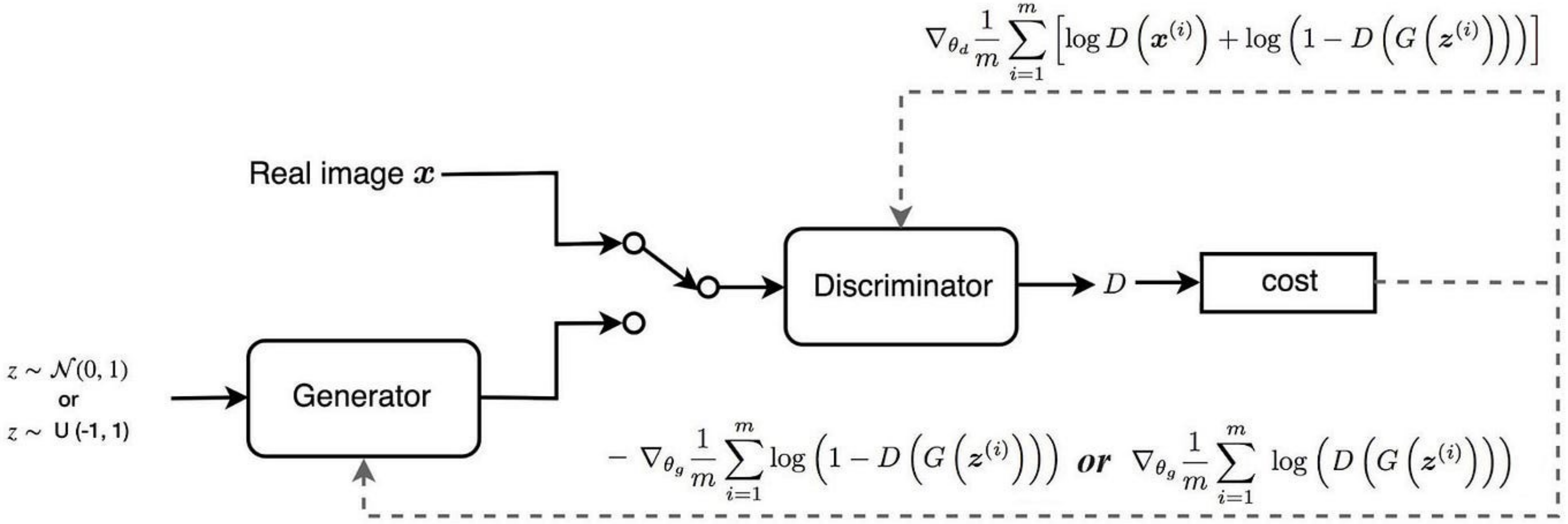
Thus, minimizing the GAN objective is equivalent to minimizing the Jensen-Shannon divergence as follows:

$$\min_G V(G, D^*) = \min_G JS(P_{\text{data}} \| P_g) + \log 4 = \min_G KL(P_{\text{data}} \| M) + KL(P_g \| M)$$

$$M(x) = \tfrac{1}{2}(P_{\text{data}}(x) + P_g(x))$$

Minimax Loss

The Standard GAN loss function can further be categorized into two parts: Discriminator loss and Generator loss. The diagram below summarizes how we train the discriminator and the generator using the corresponding gradient.



Non-Saturating GAN Loss

f-divergence

The f-divergence between probability densities p and q

$$\mathbb{D}_f(p\|q) = \mathbb{E}_{X\sim q}\left[f\left(\frac{p(X)}{q(X)}\right)\right] = \int_{\mathcal{X}} f\left(\frac{p(x)}{q(x)}\right) q(x)dx$$

where $f : (0, +\infty) \mapsto \mathcal{R}^1$ is a convex function with $f(1) = 0$.

- If $p = q$, then $\mathbb{D}_f(p\|q) = 0$.
- Jensen's inequality:

$$\mathbb{D}_f(p\|q) \geq f\left[\mathbb{E}_{X\sim q}\left(\frac{p(X)}{q(X)}\right)\right] = f(1) = 0.$$

- If $f(x)$ is strictly convex in a neighborhood of 1, then

$$\mathbb{D}_f(p\|q) = 0 \iff p = q.$$

- The KL divergence is a special case by taking $f(x) = x\log x - x + 1$.

- KL: $f(x) = x\log(x) - x + 1$.
- χ^2: $f(x) = \frac{1}{2}(x-1)^2$.
- Hellinger: $f(x) = 2(\sqrt{x} - 1)$.
- L_1: $f(x) = |x - 1|$.
- Jensen-Shannon (JS): $f(x) = x\log x - (x+1)\log\frac{x+1}{2}$.

Least Squares GAN (LSGAN)

- **Objective:** Improve stability and quality of GAN training by replacing standard binary cross-entropy loss with least squares loss.
- **Proposed by:** Mao et al. (2017) in the paper *"Least Squares Generative Adversarial Networks"*
- **Key Motivation:** Standard GANs suffer from **vanishing gradients** when the discriminator becomes too confident. Least squares loss provides **stronger gradients** and better sample quality.

In LSGAN, the **discriminator** is trained with the following **least squares loss**:

$$L_D = \frac{1}{2}\mathbb{E}_{x\sim p_{data}}[(D(x)-1)^2] + \frac{1}{2}\mathbb{E}_{z\sim p_z}[D(G(z))^2]$$

In LSGAN, the **generator** is trained to minimize:

$$L_G = \frac{1}{2}\mathbb{E}_{z\sim p_z}[(D(G(z))-1)^2]$$

Minimizing LSGAN loss is equivalent to minimizing the Pearson χ^2 divergence.

- ❖ LSGAN minimizes the Pearson χ^2 divergence, making it more stable than standard GANs.
- ❖ Compared to Jensen-Shannon divergence (used in standard GANs), χ^2 divergence is more sensitive to small differences in distributions.
- ❖ LSGAN penalizes fake samples more aggressively, which helps avoid mode collapse.

Wasserstein GAN (WGAN)

Let Ω be a subset of $\mathbb{R}^d$. Let $\mathcal{B}_p(\Omega)$ be the set of Borel probability measures on Ω with finite pth moment. The p-Wasserstein metric is defined as

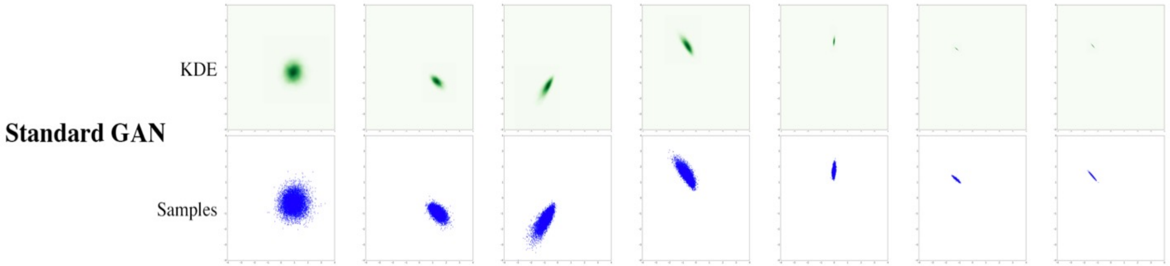
$$W_p(\mu,\nu) = \left(\inf_{\gamma \in \Gamma(\mu,\nu)} \int |x-y|^p d\gamma(x,y) \right)^{1/p}, \quad \mu \text{ and } \nu \in \mathcal{B}_p(\Omega).$$

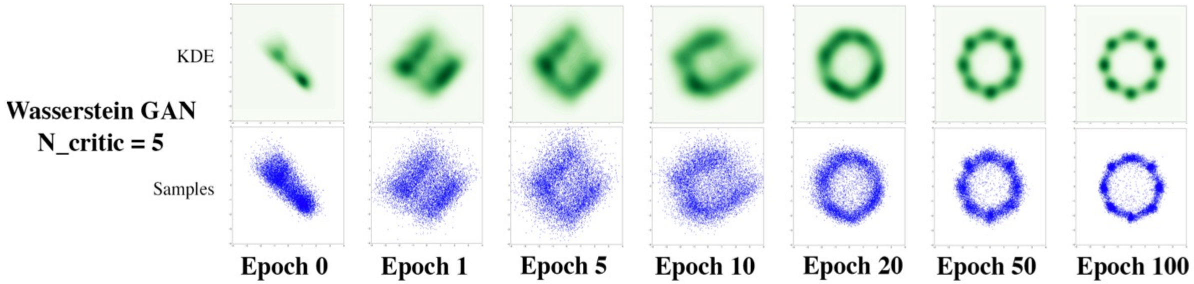
For the special case of p = 1, the p-Wasserstein metric is also known as the Monge-Rubinstein metric, or the earth mover distance.

The 1-Wasserstein metric can be expressed as (Villani, 2008),

$$W_1(\mu,\nu) = \sup_{f \in \mathcal{F}_1} \left\{ \int f(x) d\mu(x) - \int f(x) d\nu(x) \right\},$$

This expression of 1-Wasserstein metric is computationally convenient, which is used in the construction of Wasserstein generative adversarial networks (WGAN) (Arjovsky et al., 2017).





Training WGAN

Critic (Discriminator) Loss:

$$L_D = \mathbb{E}_{x \sim P_{\text{data}}}[D(x)] - \mathbb{E}_{z \sim P_z}[D(G(z))]$$

$$\nabla_{\theta_D} L_D = \nabla_{\theta_D} \left(\mathbb{E}_{x \sim P_{\text{data}}}[D(x)] - \mathbb{E}_{z \sim P_z}[D(G(z))] \right)$$

$$\theta_D \leftarrow \theta_D + \eta_D \nabla_{\theta_D} L_D$$

Methods to Enforce Lipschitz Constraint:

Weight Clipping (Original WGAN):

$$-c \leq w \leq c$$

Gradient Penalty (WGAN-GP):

$$L_{\text{GP}} = \lambda \mathbb{E}_{\hat{x} \sim P_{\hat{x}}}[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2]$$

Generator Loss:

$$L_G = -\mathbb{E}_{z \sim P_z}[D(G(z))]$$

$$\nabla_{\theta_G} L_G = -\nabla_{\theta_G} \mathbb{E}_{z \sim P_z}[D(G(z))]$$

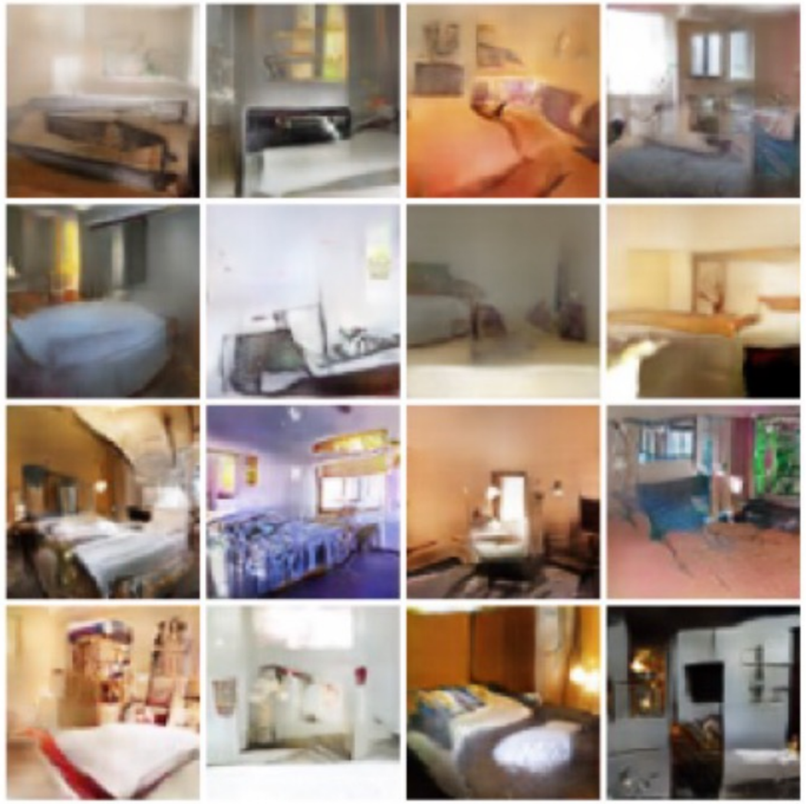
$$\theta_G \leftarrow \theta_G + \eta_G \nabla_{\theta_G} L_G$$

Training Process:

1. Update the critic D multiple times per generator update.
2. Compute Wasserstein distance using the critic’s output.
3. Update generator G to minimize the critic’s output.

WGAN vs WGAN-NP

Wasserstein GAN (WGAN)



Arjovsky, Chintala, and Bouttou, "Wasserstein GAN", 2017

WGAN with Gradient Penalty (WGAN-GP)

Gulrajani et al, "Improved Training of Wasserstein GANs", NeurIPS 2017

Progressive GAN (PGAN)

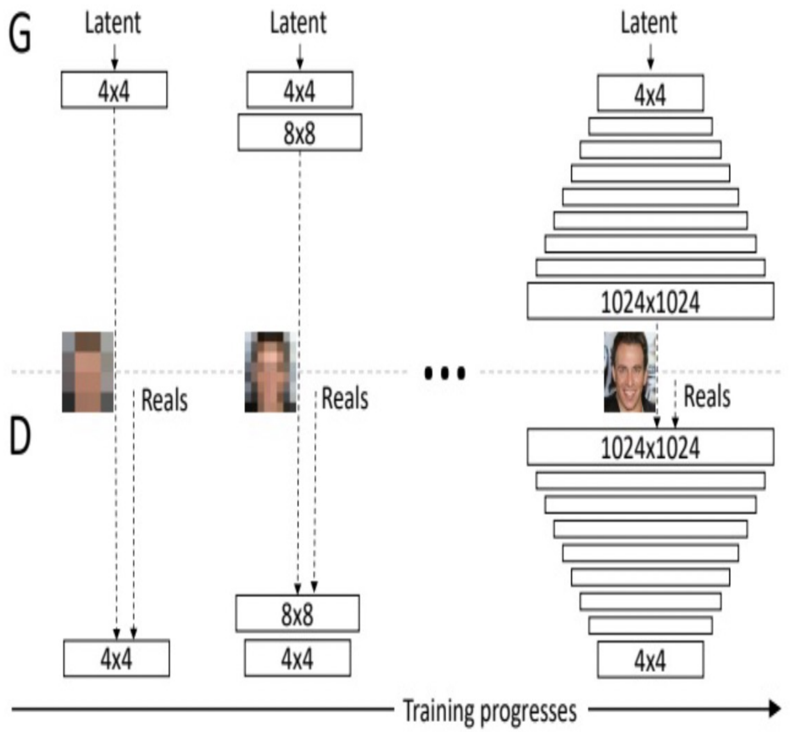
- Progressive GAN (PGAN) is a technique for progressive growth of layers in both the generator and discriminator.
- Introduced by Karras et al. (2017) for high-resolution image synthesis.
- Allows training GANs stably at resolutions up to 1024×1024.
- Works by starting small and growing larger over training iterations.

Progressive Growing Mechanism

- Training begins with low-resolution images (e.g., 4×4).
- New layers are added progressively to both generator and discriminator.
- Old layers remain trainable, allowing smooth transition.
- Uses smooth transition (fade-in layers) when adding new resolutions.

Architecture of Progressive GAN

- Generator and Discriminator start with small networks (4×4).
- New convolutional layers are added progressively to increase resolution.
- Uses skip connections to stabilize training.
- Mini-batch standard deviation is used to improve diversity.



Code: Vanilla GAN

```
import torch
import torch.nn as nn
import torchvision.models as models

# ----- Define Hyperparameters ---
lr = 0.0002       # Learning rate
z_dim = 64        # Dimensionality of the noise vector
image_dim = 28*28 # 784 for MNIST (28 x 28)
hidden_dim = 128  # Hidden layer dimensionality for both Generator and Discriminator
batch_size = 128
epochs = 50       # number of epoches

# ----- Define the Generator -----
# A fully connected (MLP) generator that takes a random noise vector z and outputs a 28x28 image (784-dimensional vector). We apply a Tanh activation to the final layer to constrain the pixel values between -1 and 1.
class Generator(nn.Module):
    def __init__(self, z_dim, hidden_dim, out_dim):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(noise_dim, hidden_dim),
            nn.ReLU(True),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(True),
            nn.Linear(hidden_dim, out_dim),
            nn.Tanh()
        )

    def forward(self, x):
        return self.net(x)
```

Code: Vanilla GAN

```
import torch
import torch.nn as nn
import torchvision.models as models

# ----- Define the Discriminator -----
# A fully connected (MLP) discriminator that takes a 784-dimensional vector (flattened 28x28 image) and outputs a single probability (real vs. fake). We apply a Sigmoid at the end to interpret the output as a probability.
class Discriminator(nn.Module):
    def __init__(self, in_dim, hidden_dim):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim, hidden_dim),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.net(x)

# -----  Instantiate Model and Optimizers  -----
# Initialize generator and discriminator
gen = Generator(z_dim, hidden_dim, image_dim).to(device)
disc = Discriminator(image_dim, hidden_dim).to(device)
criterion = nn.BCELoss() # Binary Cross Entropy loss
# Optimizers (use Adam for both)
optimizer_gen = optim.Adam(gen.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_disc = optim.Adam(disc.parameters(), lr=lr, betas=(0.5, 0.999))
```

Controllable Generation and Conditional GAN

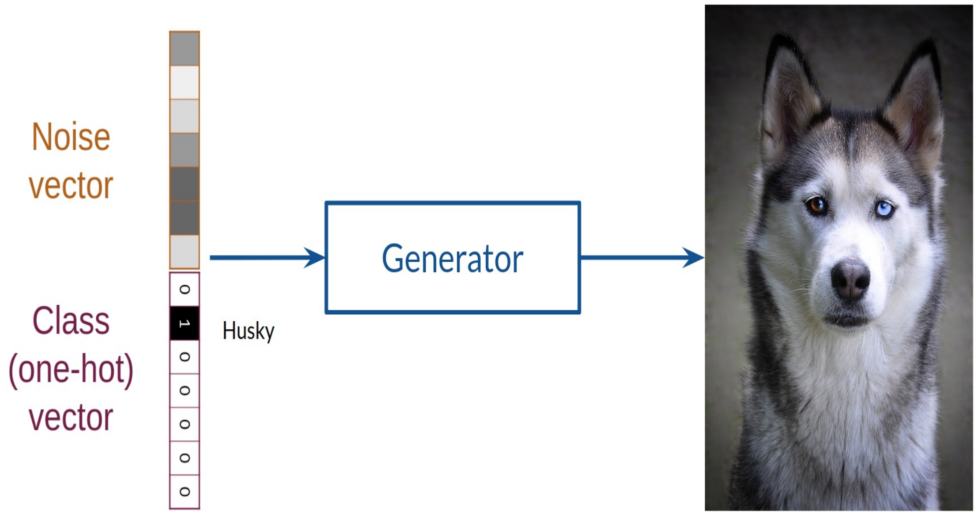


Illustration of example to generate a breed of dog (source)

Most of the practical applications require the ability to sample a conditional distribution, like:

- Next frame prediction.
- “in-painting”,
- segmentation,
- style transfer.

This would in particular address some of the shortcomings of unconditional GANs.

- CGAN is a supervised extension of GANs where the generator and discriminator receive additional information.
- Allows generating samples based on specific conditions.

CGAN

The CGAN proposed by Mirza and Osindero (2014) consists of parameterizing both G and D by a conditioning quantity Y.

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

This adds semantic meaning to latent space manifold and provides more control in the types of output generated by the generator.

Training Algorithm:

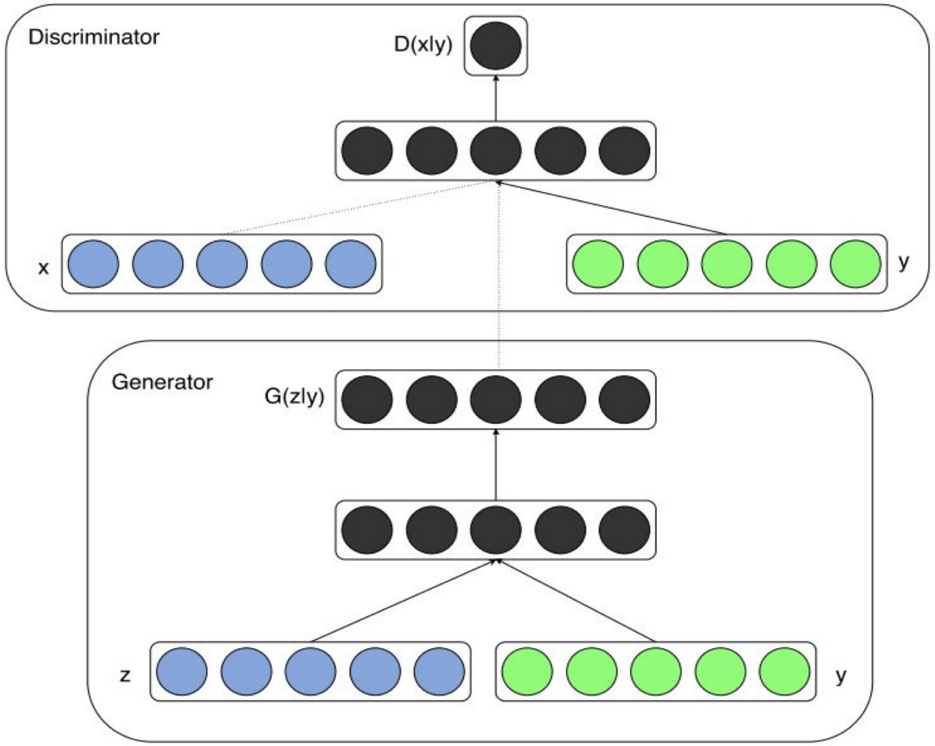
1. Initialize generator G, discriminator D, and dataset.
2. For each iteration:
   - Sample real data $(x, y) \sim p_{data}(x, y)$.
   - Generate fake samples $G(z, y)$ with $z \sim p_z(z)$.
   - Update D by minimizing:

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

   - Update G by minimizing:

     $$L_G = -\mathbb{E}_{z\sim p_z,y\sim p_y}[\log D(G(z,y),y)]$$

3. Repeat until convergence.

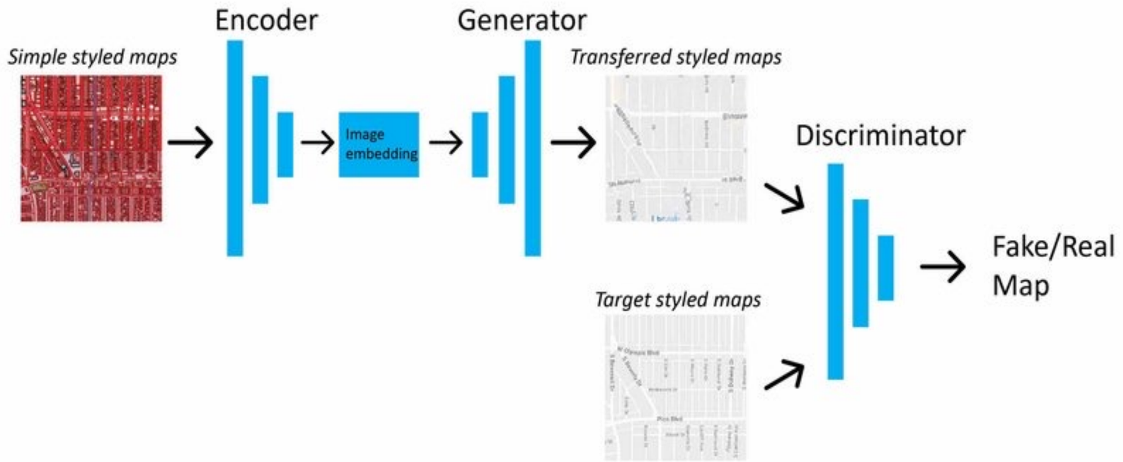


Pix2Pix

- Pix2Pix is a supervised image-to-image translation model based on cGANs.
- Introduced in Isola et al. (2017) for tasks like sketch-to-photo, satellite-to-map, and more.

Pix2Pix Objective: $G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G, D) + \lambda \mathcal{L}_{L1}(G)$ $\quad \mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}\|y - G(x,z)\|_1$

$$\mathcal{L}_{cGAN}(G, D) = \mathbb{E}_{x,y}[\log D(x,y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x,z)))]$$



Training Algorithm

1. Initialize generator G and discriminator D.
2. For each training step:
   - Sample real image pairs $(x, y) \sim p_{data}(x, y)$.
   - Generate fake image $G(x, z)$.
   - Update D by minimizing:

     $$L_D = -\mathbb{E}[\log D(x,y)] - \mathbb{E}[\log(1 - D(x, G(x,z)))]$$

   - Update G by minimizing:

     $$L_G = -\mathbb{E}[\log D(x, G(x,z))] + \lambda\|y - G(x,z)\|_1$$

3. Repeat until convergence.

Pix2Pix Generator: U-Net

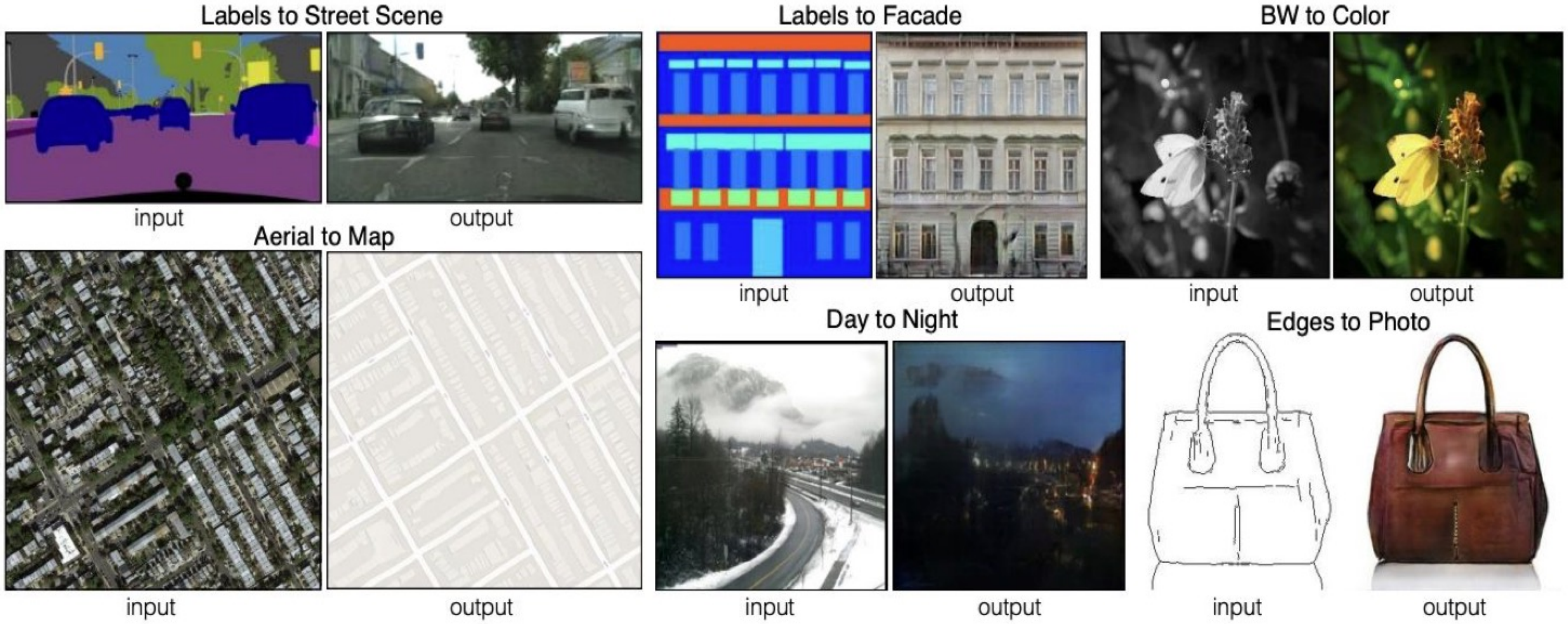
- The generator is based on a U-Net architecture, using an encoder-decoder structure with skip connections.
- The encoder extracts deep features while the decoder reconstructs the image.
- Skip connections help preserve fine-grained details.

Pix2Pix Discriminator: PatchGAN

- Uses a convolutional PatchGAN discriminator instead of a full-image classifier.
- PatchGAN classifies small image patches instead of the entire image.
- Helps focus on local texture realism and prevents blurriness.

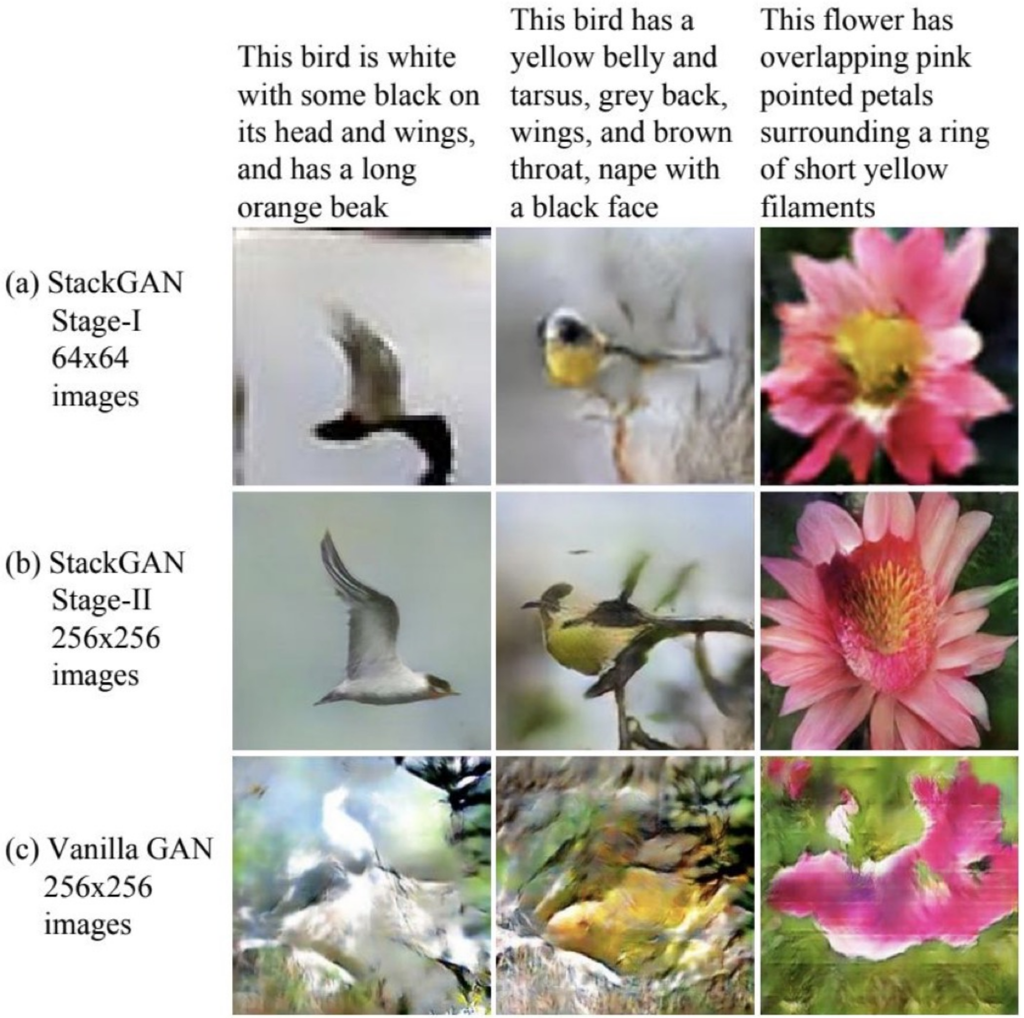
Application of CGANs

Image-to-image translation



https://arxiv.org/pdf/1611.07004.pdf

Application of CGANs



Text to image synthesis

Comparison of StackGAN (stacked conditional GAN) and a one-stage GAN for generating 256×256 images. (a) Given text descriptions, Stage-I of StackGAN sketches rough shapes and basic colors of objects, yielding low-resolution images. (b) Stage-II of StackGAN takes Stage-I results and text descriptions as inputs, and generates high-resolution images with photo-realistic details. (c) Results by a vanilla 256×256 GAN which simply adds more upsampling layers to state-of-the-art GAN-INT-CLS [26]. It is unable to generate any plausible images of 256×256 resolution.

http://arxiv.org/pdf/1612.03242

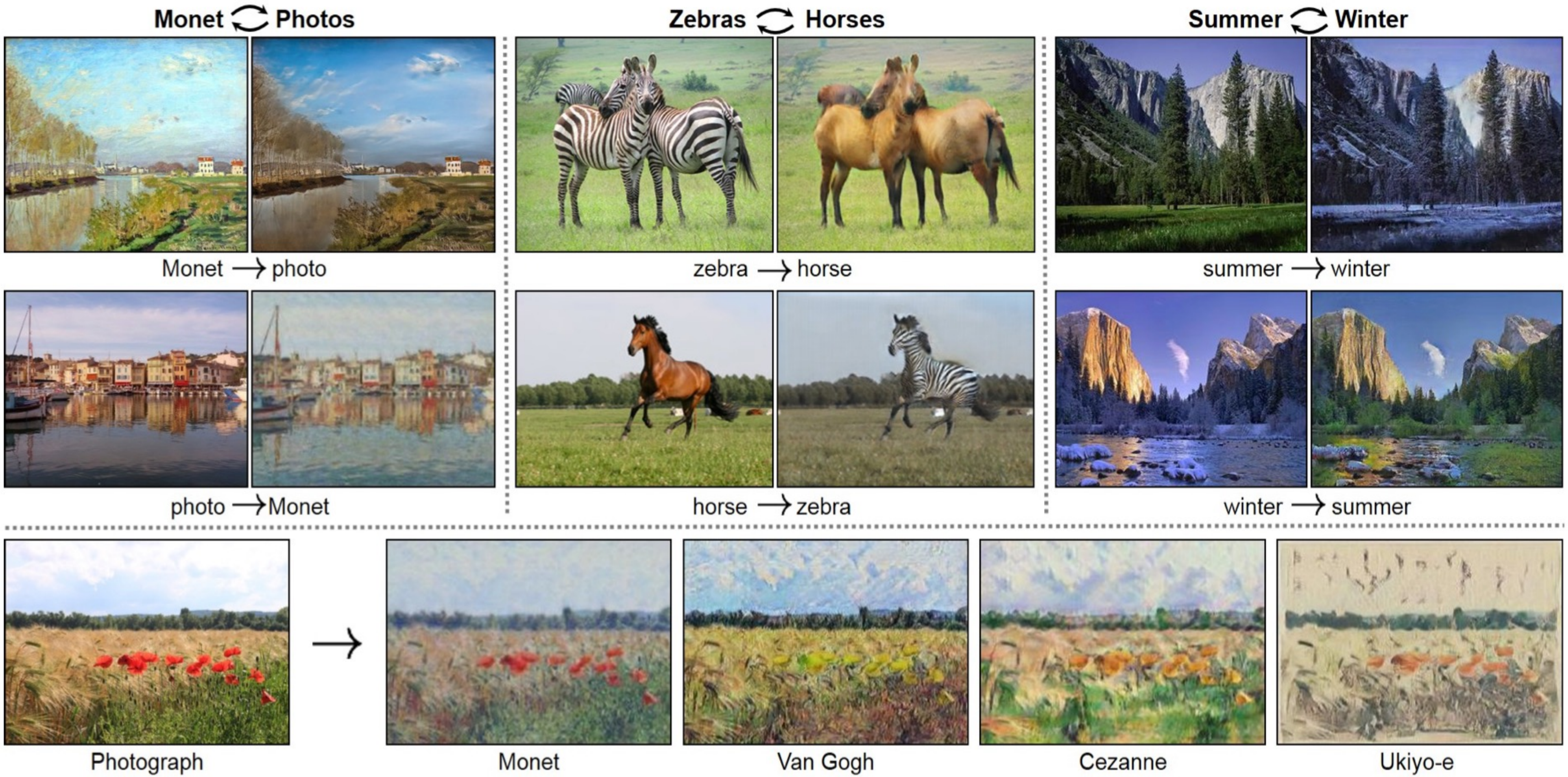
Code: Conditional GAN

```
import torch
import torch.nn as nn

# ----- Define Hyperparameters ---
lr = 0.0002        # Learning rate
z_dim = 64         # Dimensionality of the noise vector
image_dim = 28*28 # 784 for MNIST (28 x 28)
hidden_dim = 128   # Hidden layer dimensionality for both Generator and Discriminator
batch_size = 128
epochs = 50        # number of epoches
# ----- Define the Generator -----
# We concatenate [z, label_onehot] into a single vector of size z_dim + label_dim before passing through an MLP. The output is a flattened 28x28 image (size 784), which we squish to [-1,1] using Tanh.
class Generator(nn.Module):
    def __init__(self, z_dim, label_dim, hidden_dim, out_dim):
        super(Generator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(noise_dim, hidden_dim),
            nn.ReLU(True),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(True),
            nn.Linear(hidden_dim, out_dim),
            nn.Tanh()
        )
    def forward(self, x):
        # labels: (batch_size, label_dim)
        # z: (batch_size, z_dim)
        x = torch.cat([z, labels], dim=1)  # Concatenate noise + label
        return self.net(x)
```

Code: Conditional GAN

```
import torch
import torch.nn as nn

# ----- Define the Discriminator -----
# We concatenate [image, label_onehot] into a single vector of size image_size + label_dim before passing through an
MLP. The final output is a single probability (real or fake), obtained via Sigmoid.
class Discriminator(nn.Module):
    def __init__(self, in_dim, label_dim, hidden_dim):
        super(Discriminator, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(in_dim + label_dim, hidden_dim),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_dim, hidden_dim),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(hidden_dim, 1),
            nn.Sigmoid()
        )
    def forward(self, x, labels):
        # x: (batch_size, image_size)
        # labels: (batch_size, label_dim)
        x = torch.cat([x, labels], dim=1)  # Concatenate image + label
        return self.net(x)

# -----  Instantiate Model and Optimizers  -----
gen = Generator(z_dim, label_dim, hidden_dim, image_size).to(device)
disc = Discriminator(image_size, label_dim, hidden_dim).to(device)
criterion = nn.BCELoss()
optimizer_gen = optim.Adam(gen.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_disc = optim.Adam(disc.parameters(), lr=lr, betas=(0.5, 0.999))
```

Cycle GAN to Transfer Image Domains



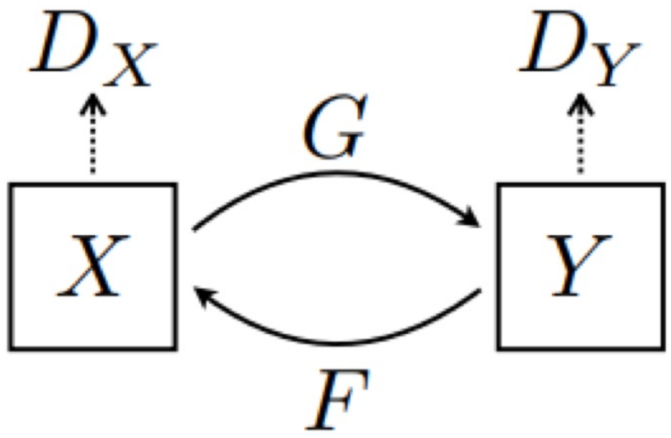
Cycle GAN

- CycleGAN is an unsupervised image-to-image translation method.
- Unlike Pix2Pix, it does not require paired data.
- Uses two generators and two discriminators for learning mappings between two domains.
- Useful for photo enhancement, style transfer, and domain adaptation.

❖ Convert an image from one representation to another.

❖ Capture characteristics of one image domain and figure out how these characteristics could be translated into the other domain.

❖ Two mapping G : X → Y and F : Y → X. Two discriminators: D_X and D_Y.



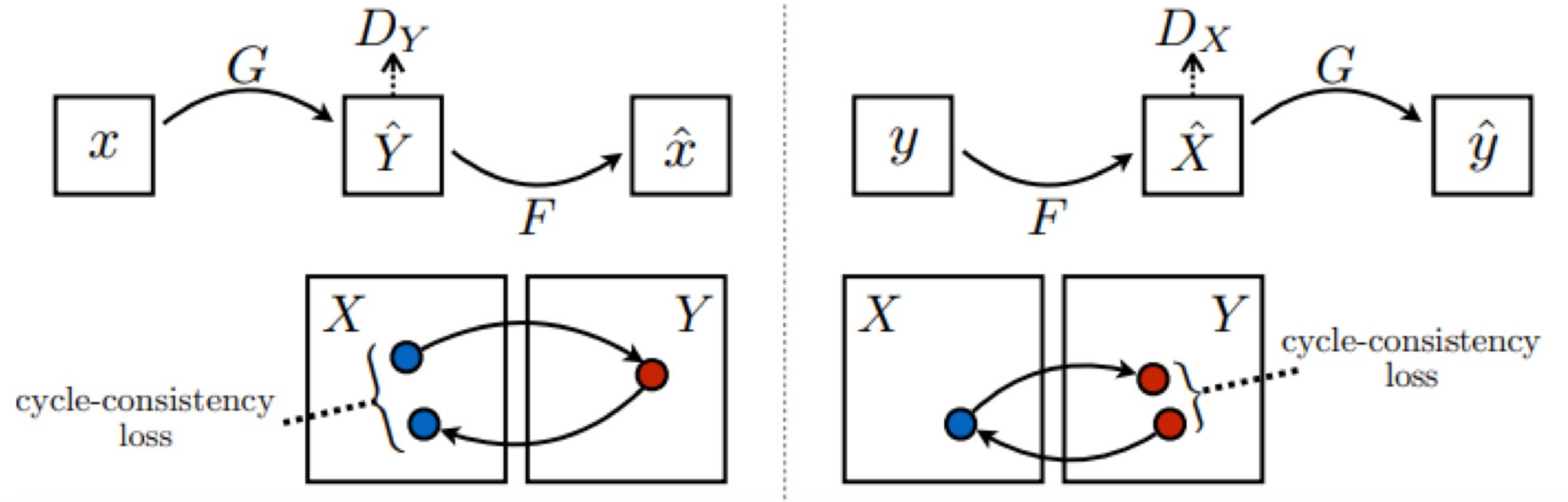
Encourage G to generate images similar to images in domain Y and D_Y to distinguish G(x) from y.

$$L_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y\sim p_{data}(y)}[log D_Y(y)] + \mathbb{E}_{x\sim p_{data}(x)}[log(1 - D_Y(G(x)))]$$

Similarly, for F, we also have $L_{GAN}(F, D_X, X, Y)$.

Consistency Loss

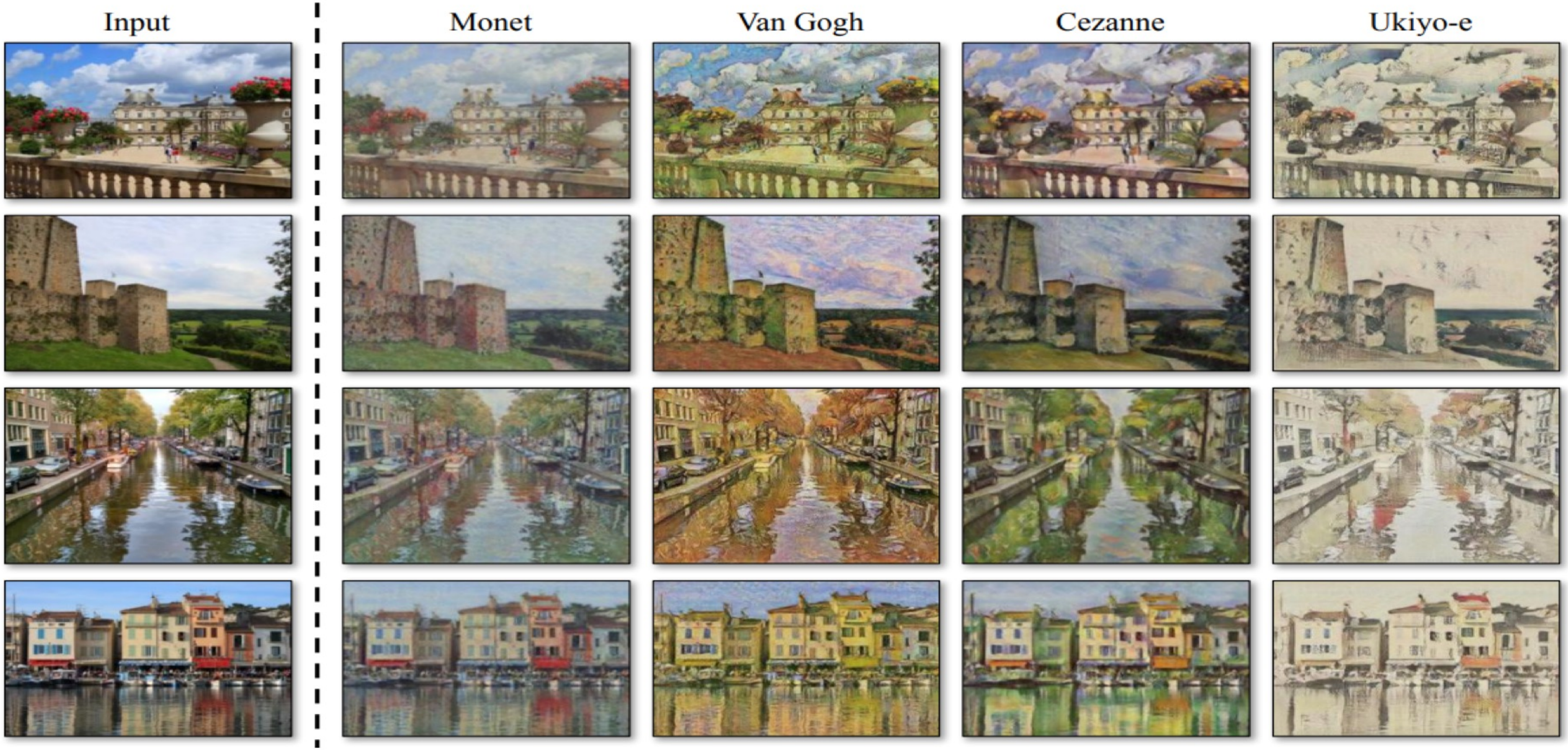
- Need extra regularization to make sure mapping function is cycle-consistent, (capable of mapping the input images to any subsets of images in the target domain), i.e., x → G(x) → F(G(x)) ≈ x.
- Cycle consistency loss: $L_{cyc}(G, F) = \mathbb{E}_{x\sim p_{data}(x)}\left[\left\|F(G(x)) - x\right\|_1\right] + \mathbb{E}_{y\sim p_{data}(y)}\left[\left\|G(F(y)) - y\right\|_1\right]$
- The full objective: $L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, X, Y)$.



Cycle GAN Example Result

Jun-Yan Zhu*, Taesung Park*, Phillip Isola, and Alexei A. Efros. ”Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks”, in IEEE International Conference on Computer Vision (ICCV), 2017

Cycle GAN Example Result



Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros Image-to-Image Translation with Conditional Adversarial Networks, CVPR, 2017

Tips to improve GAN performance

- Change the cost function for a better optimization goal.
- Add additional penalties to the cost function to enforce constraints.
- Avoid overconfidence and overfitting.
- Better ways of optimizing the model.
- Add labels (Conditional GAN).
- More details and other implementation tips: https://towardsdatascience.com/gan-ways-to-improve-gan-performance-acf37f9f59b

Code: Cycle GAN

```
# The key idea is to learn two translation mappings
between two domains X and Y without requiring paired
examples.
# ----- Define Residual Blocks -----
class ResidualBlock(nn.Module):
    """Residual Block with instance normalization."""
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.block = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(channels, channels, kernel_size=3,
stride=1),
            nn.InstanceNorm2d(channels),
            nn.ReLU(True),

            nn.ReflectionPad2d(1),
            nn.Conv2d(channels, channels, kernel_size=3,
stride=1),
            nn.InstanceNorm2d(channels)
        )
    def forward(self, x):
        return x + self.block(x)

# - Define ResNet-based Generators (G: X → Y, F: Y → X)--
class GeneratorResNet(nn.Module):
    """
    Generator that transforms input images (domain X to
domain Y or vice versa).
    Uses several downsampling layers, residual blocks, and
upsampling layers.
    """
```

```
def __init__(self, in_channels, out_channels, n_res_blocks=6,
ngf=64):
        super(GeneratorResNet, self).__init__()
        # Initial convolution block
        model = [nn.ReflectionPad2d(3),
            nn.Conv2d(in_channels, ngf, kernel_size=7, stride=1),
            nn.InstanceNorm2d(ngf),
            nn.ReLU(True)]
        # Downsampling
        curr_dim = ngf
        for _ in range(2):
            model += [nn.Conv2d(curr_dim, curr_dim*2, kernel_size=
stride=2, padding=1),
                nn.InstanceNorm2d(curr_dim*2),
                nn.ReLU(True)]
            curr_dim *= 2
        # Residual blocks
        for _ in range(n_res_blocks):
            model += [ResidualBlock(curr_dim)]
        # Upsampling
        for _ in range(2):
            model += [nn.ConvTranspose2d(curr_dim, curr_dim//2,
kernel_size=3, stride=2, padding=1, output_padding=1),
nn.InstanceNorm2d(curr_dim//2),nn.ReLU(True)]
            curr_dim //= 2
        # Output layer
        model += [nn.ReflectionPad2d(3), nn.Conv2d(curr_dim,
out_channels, kernel_size=7, stride=1), nn.Tanh()]
        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)
```

Code: Cycle GAN

```
# ----- Define the Discriminator -----
class Discriminator(nn.Module):
    """
    PatchGAN discriminator: tries to classify each NxN patch in the image
    as real or fake. Output is a feature map of "realness" scores.
    """
    def __init__(self, in_channels=3, ndf=64):
        super(Discriminator, self).__init__()
        # A small patch-based ConvNet
        model = [
            nn.Conv2d(in_channels, ndf, kernel_size=4, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf, ndf*2, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(ndf*2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf*2, ndf*4, kernel_size=4, stride=2, padding=1),
            nn.InstanceNorm2d(ndf*4),
            nn.LeakyReLU(0.2, inplace=True),

            # Last convolution
            nn.Conv2d(ndf*4, 1, kernel_size=4, stride=1, padding=1)
            # No Sigmoid here, we use BCEWithLogitsLoss
        ]
        self.model = nn.Sequential(*model)

    def forward(self, x):
        return self.model(x)
```

Content

Imaging Synthesis

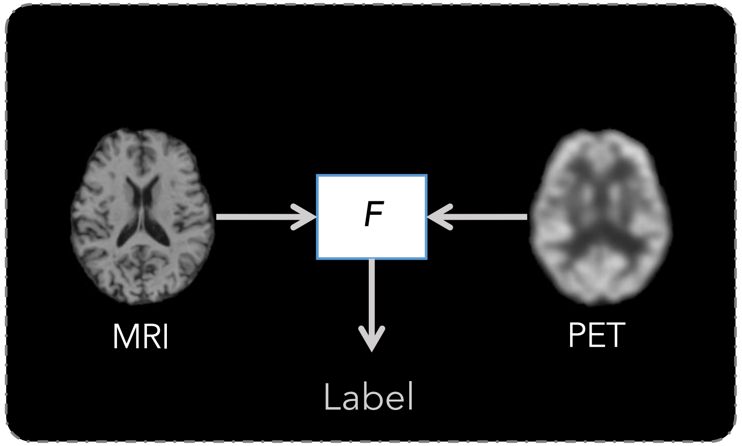
Cross-Modal Image Synthesis via Deep Learning

Y. Pan, M. Liu, C. Lian, Y. Xia, and D. Shen. *MICCAI*, 2019
Y. Pan, M. Liu, C. Lian, T. Zhou, Y. Xia, and D. Shen. *MICCAI*, 2018
Y. Pan, M. Liu, C. Lian, L. Yue, S. Xiao, Y. Xia, and D. Shen. *ISMRM*, 2019

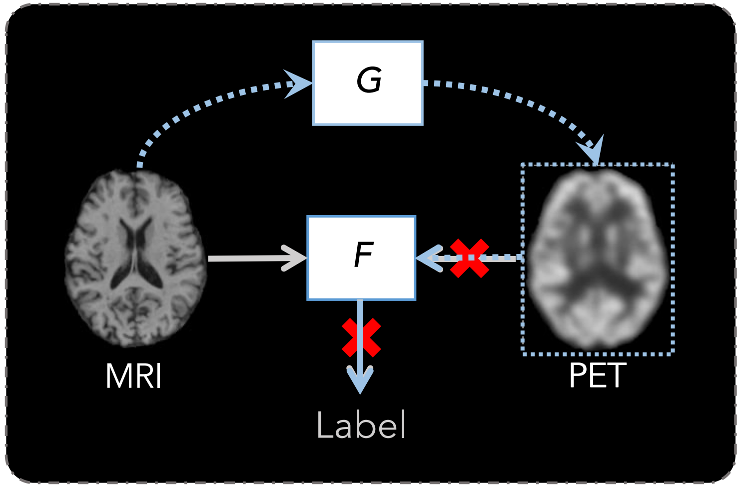
Cross-Modal Image Synthesis for Diagnosis

- Multi-modal imaging data for disease diagnosis (e.g., MRI and PET)
  - Providing complementary information of the brain

    Subjects usually have incomplete multi-modal data



Diagnosis with complete multi-modal images



Diagnosis with incomplete multi-modal images

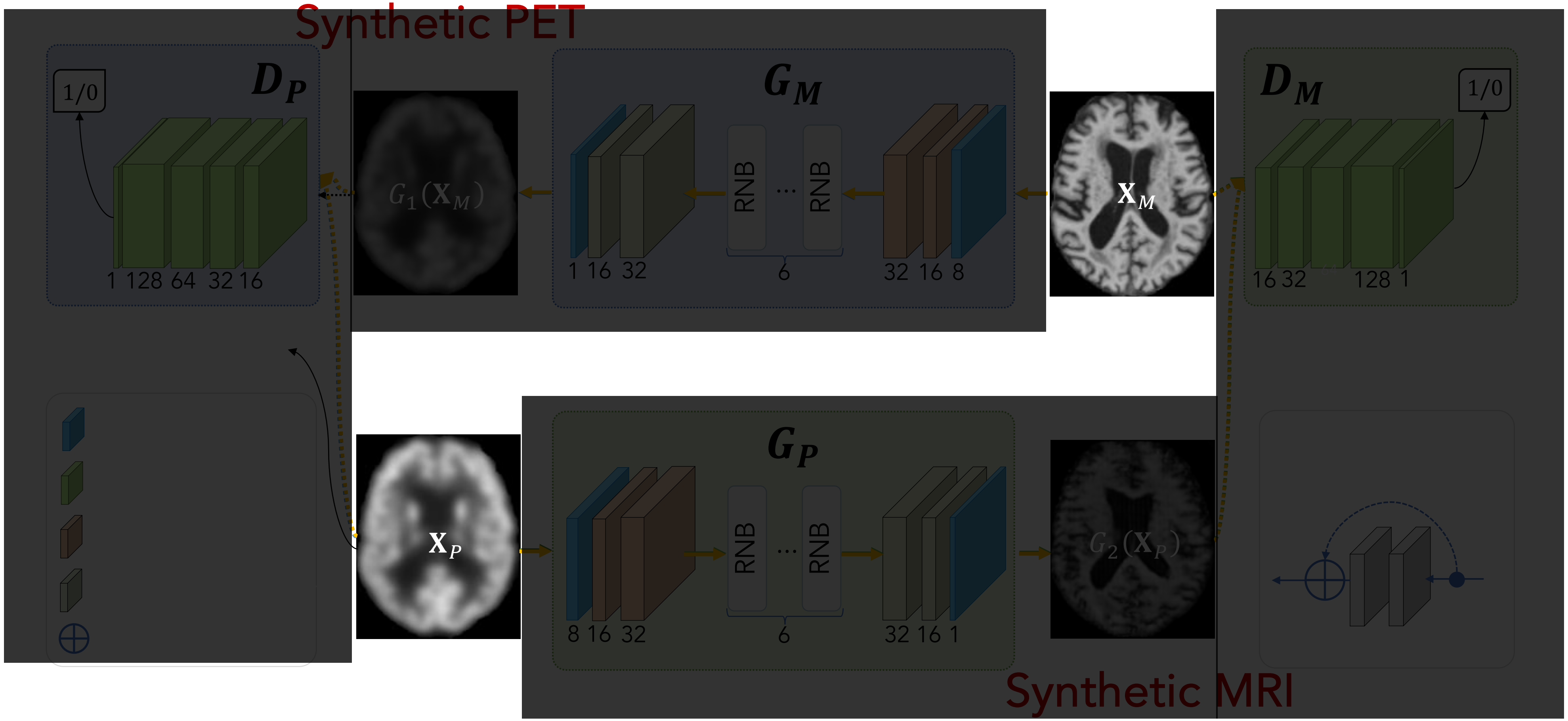
Y. Pan, M. Liu, C. Lian, Y. Xia, and D. Shen. *MICCAI*, 2019

Y. Pan, M. Liu, C. Lian, T. Zhou, Y. Xia, and D. Shen. *MICCAI*, 2018

Y. Pan, M. Liu, C. Lian, L. Yue, S. Xiao, Y. Xia, and D. Shen. *ISMRM*, 2019

Cross-Modality Image Synthesis

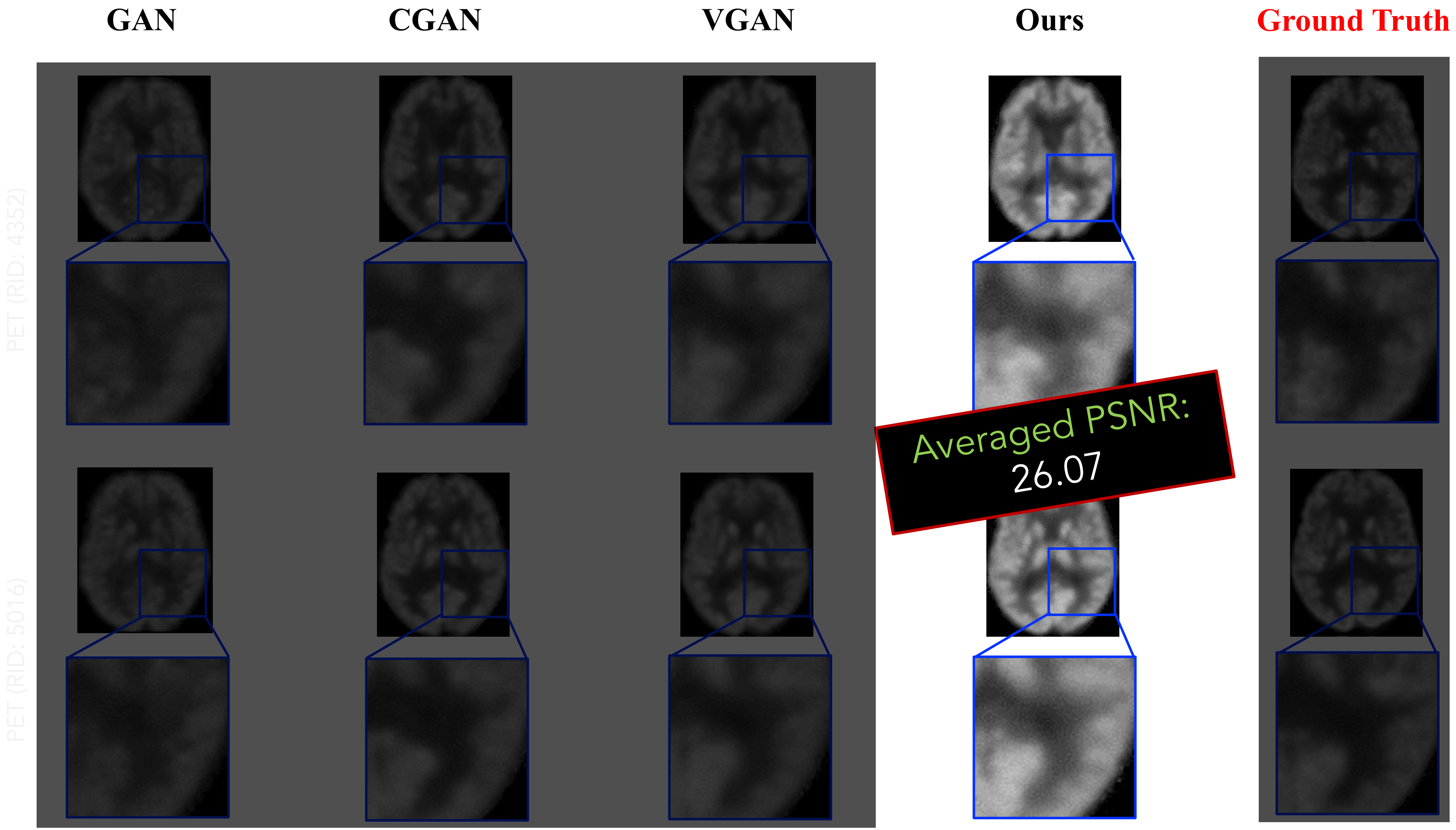
- Generating missing PET/MRI scans for diagnosis



Y. Pan, M. Liu, C. Lian, T. Zhou, Y. Xia, and D. Shen. *MICCAI,* 2018

Y. Pan, M. Liu, C. Lian, L. Yue, S. Xiao, Y. Xia, and D. Shen. *ISMRM*, 2019

Synthetic PET Scans

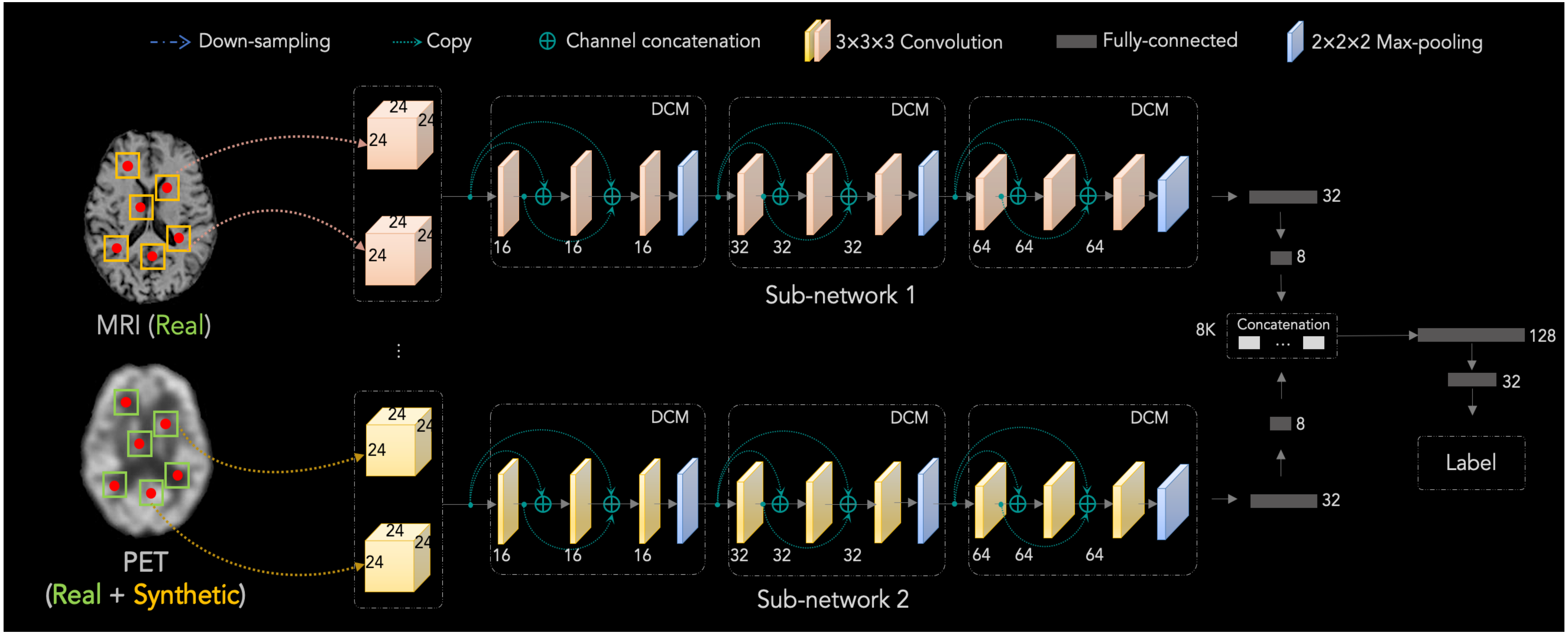


Y. Pan, M. Liu, C. Lian, T. Zhou, Y. Xia, and D. Shen. *MICCAI*, 2018
Y. Pan, M. Liu, C. Lian, L. Yue, S. Xiao, Y. Xia, and D. Shen. *ISMRM*, 2019

PSNR: Peak signal-to-noise ratio

Multi-Modal Classification

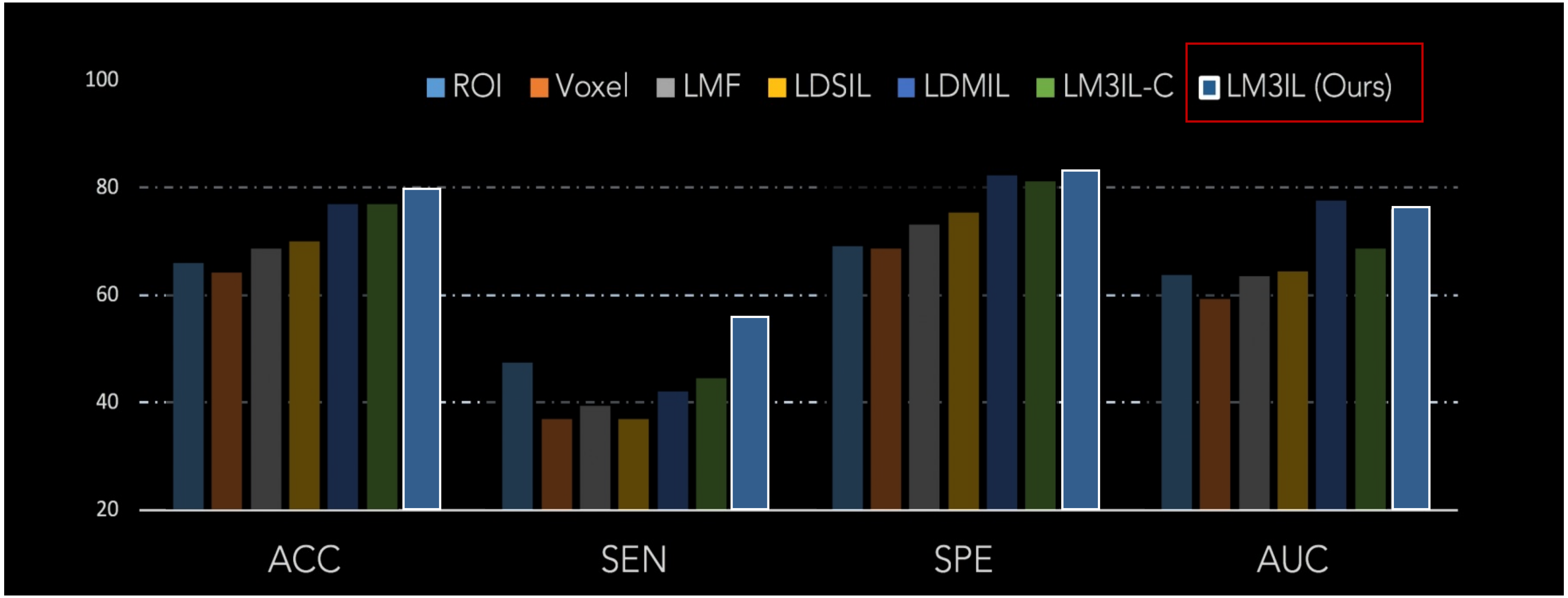
- Landmark-based multi-scale network for classification



Y. Pan, M. Liu, C. Lian, T. Zhou, Y. Xia, and D. Shen. *MICCAI*, 2018
Y. Pan, M. Liu, C. Lian, L. Yue, S. Xiao, Y. Xia, and D. Shen. *ISMRM*, 2019

Experiments

- Results of pMCI vs. sMCI with complete MRI and PET (after imputation)



Y. Pan, M. Liu, C. Lian, T. Zhou, Y. Xia, and D. Shen. *MICCAI*, 2018
Y. Pan, M. Liu, C. Lian, L. Yue, S. Xiao, Y. Xia, and D. Shen. *ISMRM*, 2019

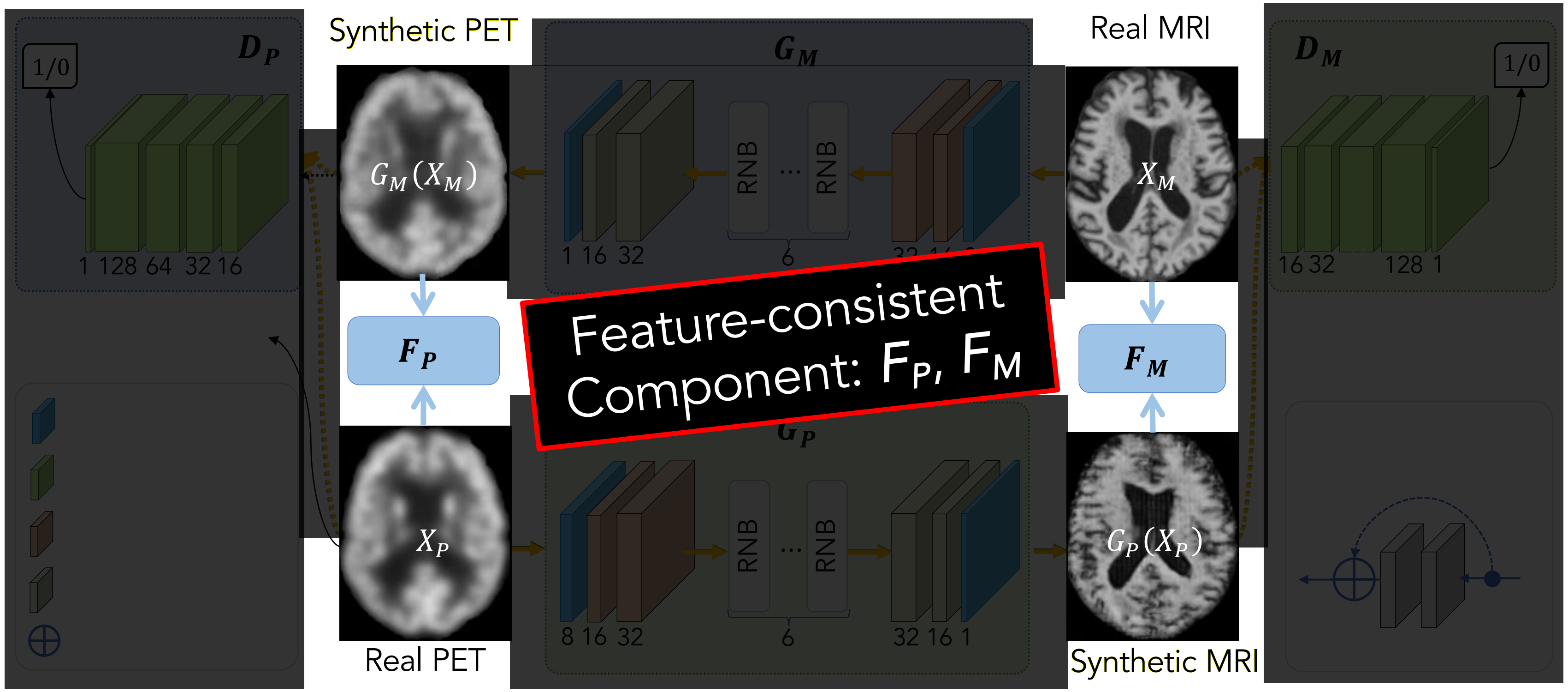
Imaging Synthesis

Diagnosis-Oriented GAN for PET/MRI Construction

Y. Pan, M. Liu, Y. Xia, and D. Shen. *IEEE Trans. Pattern Analysis and Artificial Intelligence, 2022*
Y. Liu, L. Yue, S. Xiao, W. Yang, D. Shen, M. Liu. *Medical Image Analysis, 2022*

Diagnosis-Oriented PET/MRI Construction

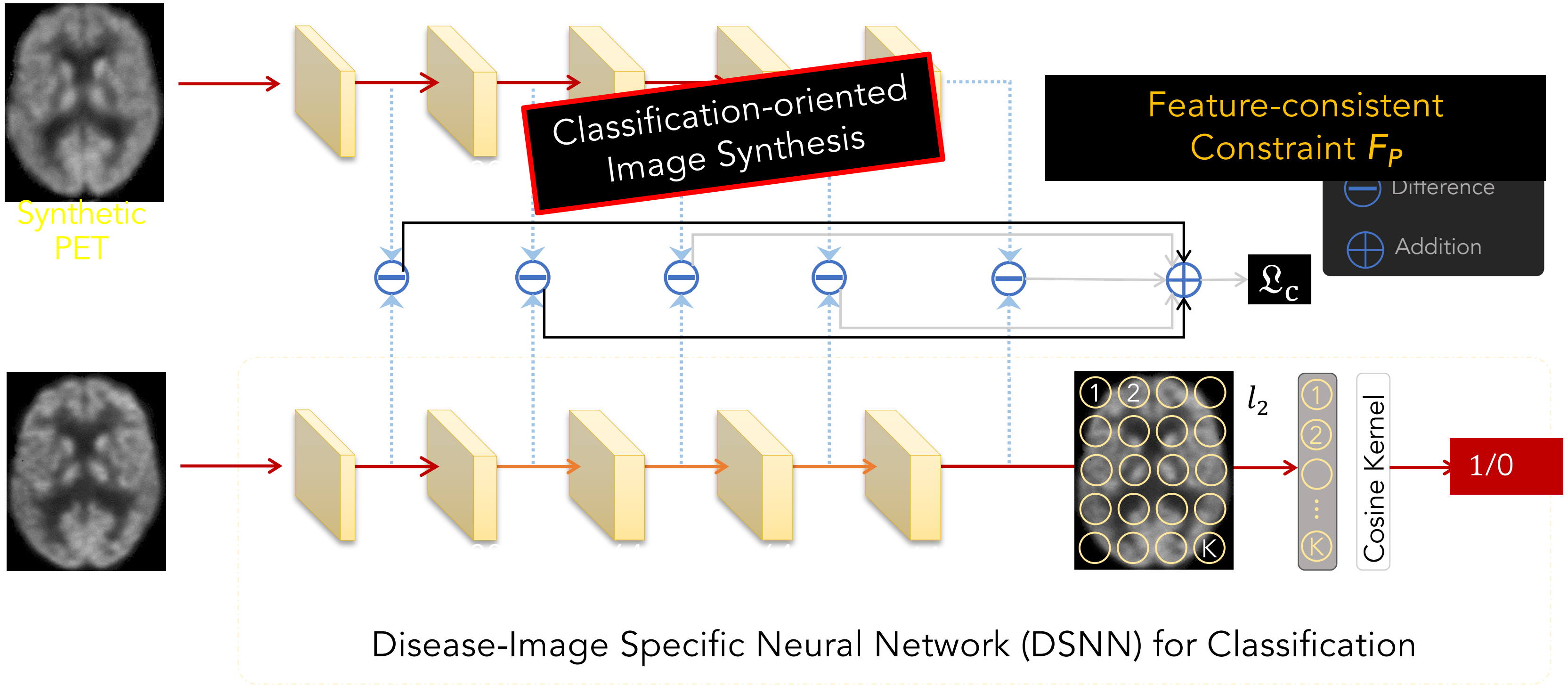
- Generating diagnosis-oriented PET/MRI scans



Y. Pan, M. Liu, Y. Xia, and D. Shen. *IEEE Trans. Pattern Analysis and Artificial Intelligence, 2022*

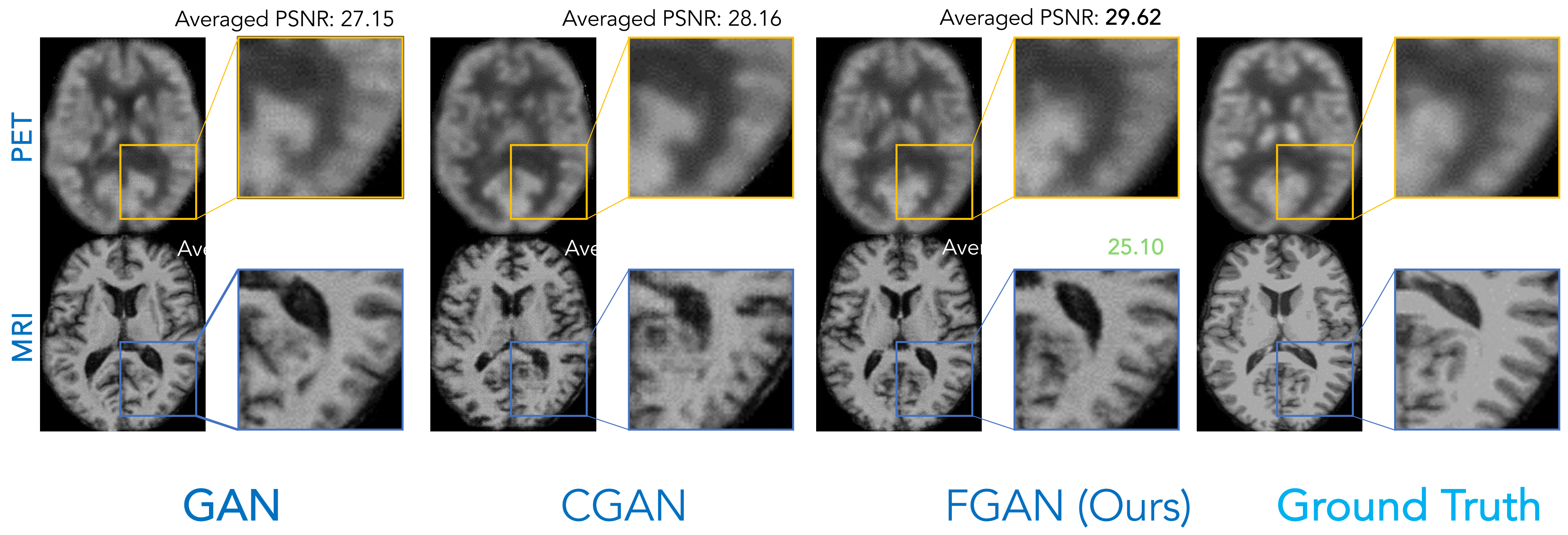
Feature-Consistent Component

- Joint classification-oriented image synthesis and classifier training



Y. Pan, M. Liu, Y. Xia, and D. Shen. *IEEE Trans. Pattern Analysis and Artificial Intelligence, 2022*

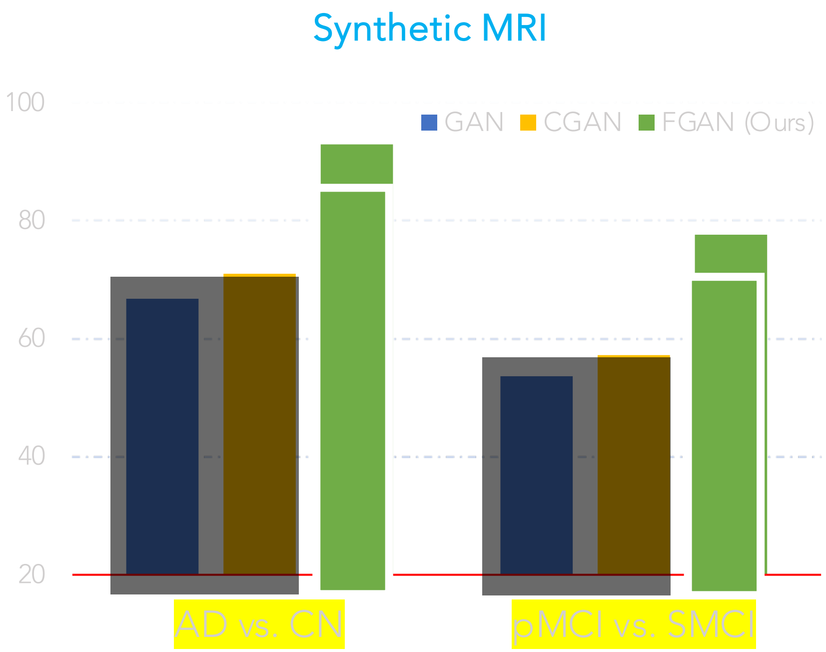
Results of Image Synthesis

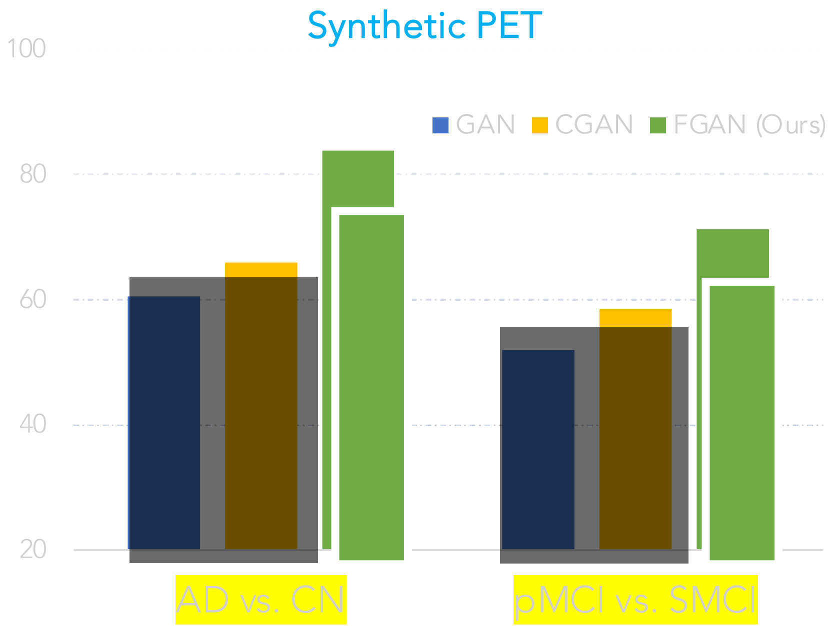


Y. Pan, M. Liu, Y. Xia, and D. Shen. *IEEE Trans. Pattern Analysis and Artificial Intelligence, 2022*

Results of Image Synthesis

- Classification results using synthetic MRI and PET scans

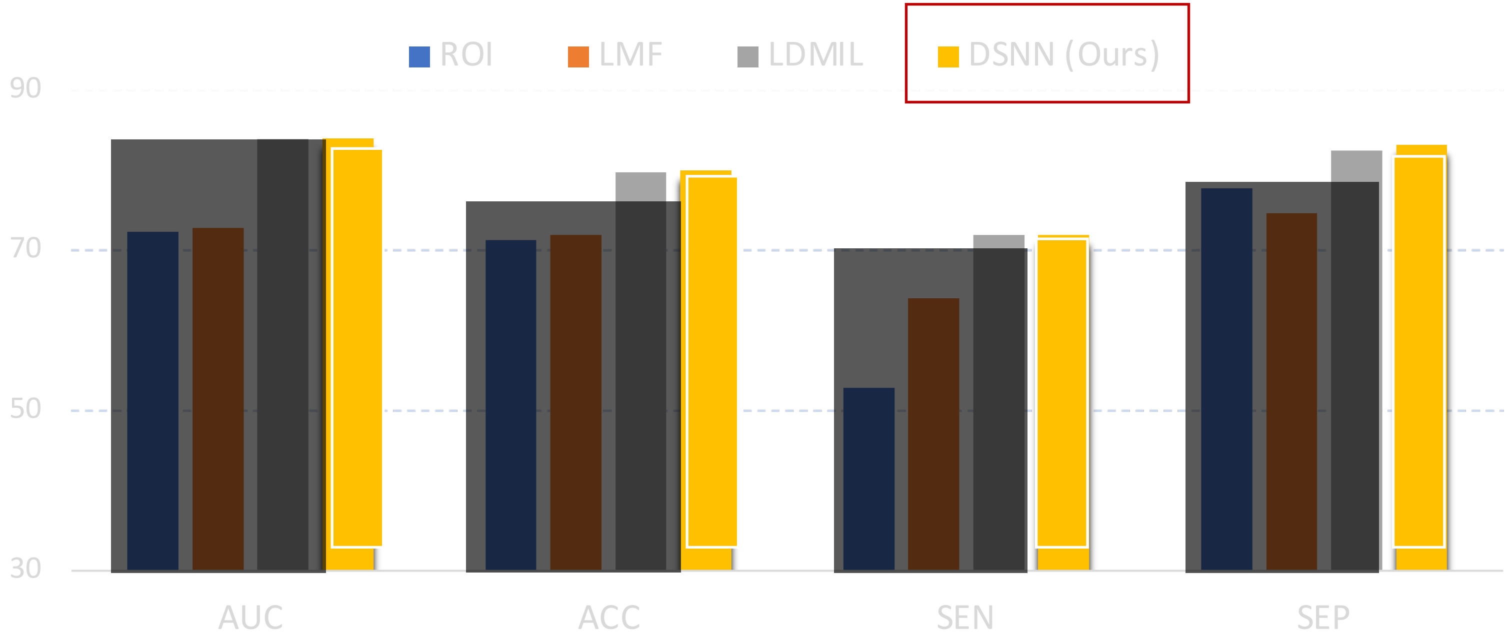




Y. Pan, M. Liu, Y. Xia, and D. Shen. *IEEE Trans. Pattern Analysis and Artificial Intelligence, 2022*

Results of Classification

- MCI conversion prediction with complete MRI and PET (after imputation)



Generating task-oriented PET scans boost performance

Y. Pan, M. Liu, Y. Xia, and D. Shen. *IEEE Trans. Pattern Analysis and Artificial Intelligence, 2022*

Content

Theoretical Properties

Lemma 1 (Noise-Outsourcing Lemma)

1. *Let (X, Y) be a random pair taking values in $\mathcal{X} \times \mathcal{Y}$ with joint distribution $P_{X,Y}$.*
2. *Suppose $\mathcal{Y}$ is a standard Borel space.*

Then there exists a random variable $\eta \sim \text{Uniform}[0,1]$ and a Borel-measurable function $G : [0,1] \times \mathcal{X} \to \mathcal{Y}$ such that η is independent of X and

$$(X, Y) = (X, G(\eta, X)) \quad \text{almost surely.} \tag{2}$$

GANs aim to approximate complex probability distributions through adversarial training.

Key theoretical questions:

- How to mathematically characterize the distribution of imaging and text datasets (e.g., ImageNet)?
- Why do we think that a specific GAN (loss function and architecture) can learn the key complexity of certain distributions?
- Can GANs converge to the true data distribution of some complex datasets?
- What distance measure is optimized in training?
- How to rigorously evaluate GANs in complex scenqrios?
- How do GANs generalize to unseen data?

- **What is the Manifold Hypothesis?**
  - High-dimensional data (e.g., images, text) often lies on a lower-dimensional manifold embedded in a higher-dimensional space.
  - Learning this low-dimensional structure is crucial for improving generative models.
- **Why Study Generative Models from this Perspective?**
  - Understanding DGMs through this lens helps explain their strengths and weaknesses.
  - Provides insights into why certain models (e.g., diffusion models, GANs) outperform others (e.g., VAEs, normalizing flows)

References

Aggarwal, A., Mittal, M., & Battineni, G. (2021). Generative adversarial network: An overview of theory and applications. *International Journal of Information Management Data Insights*, *1*(1), 100004.

Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein generative adversarial networks. In International Conference on Machine Learning.

Brophy, E., Wang, Z., She, Q., & Ward, T. (2023). Generative adversarial networks in time series: A systematic literature review. *ACM Computing Surveys*, *55*(10), 1-31.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B., & Bharath, A. A. (2018). Generative adversarial networks: An overview. *IEEE signal processing magazine*, *35*(1), 53-65.

Loaiza-Ganem, Gabriel, Brendan Leigh Ross, Rasa Hosseinzadeh, Anthony L. Caterini, and Jesse C. Cresswell. "Deep generative models through the lens of the manifold hypothesis: A survey and new connections." *arXiv preprint arXiv:2404.02954* (2024).

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In Advances in neural information processing systems, pages 2672{2680.

Gui, J., Sun, Z., Wen, Y., Tao, D., & Ye, J. (2021). A review on generative adversarial networks: Algorithms, theory, and applications. *IEEE transactions on knowledge and data engineering*, *35*(4), 3313-3332.

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). Improved training of wasserstein gans. arXiv:1704.00028.

Kingma, D.P. and Welling, M. (2019). An Introduction to variational autoencoders. Foundations and Trends® in Machine Learning 12 (4), 307-392

Mirza, Mehdi, and Simon Osindero. "Conditional generative adversarial nets." *arXiv preprint arXiv:1411.1784* (2014).

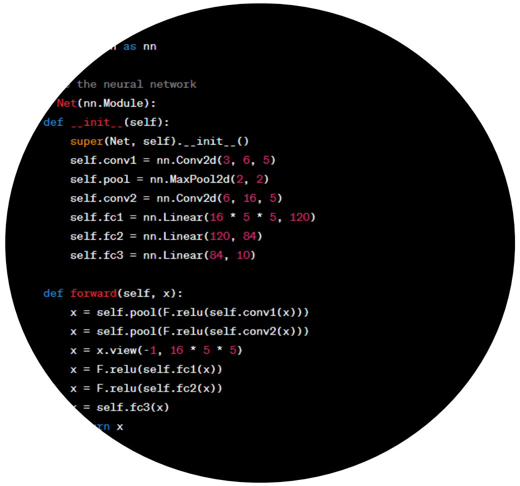
Pan, Z., Yu, W., Yi, X., Khan, A., Yuan, F., & Zheng, Y. (2019). Recent progress on generative adversarial networks (GANs): A survey. *IEEE access*, *7*, 36322-36333.

Yi, Xin, Ekta Walia, and Paul Babyn. "Generative adversarial network in medical imaging: A review." *Medical image analysis* 58 (2019): 101552.

Xu, Lei, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. "Modeling tabular data using conditional gan." *Advances in neural information processing systems* 32 (2019).

Zhou, X., Jiao, Y., Liu, J., and Huang, J. (2023). A deep generative learning approach to conditional sampling. Journal of the. American Statistical Association, 118(543):1837-1848.

How to succeed in this course?



Practice

Explore

Visualize

Discuss

Ask