Flow-based Deep Generative Models

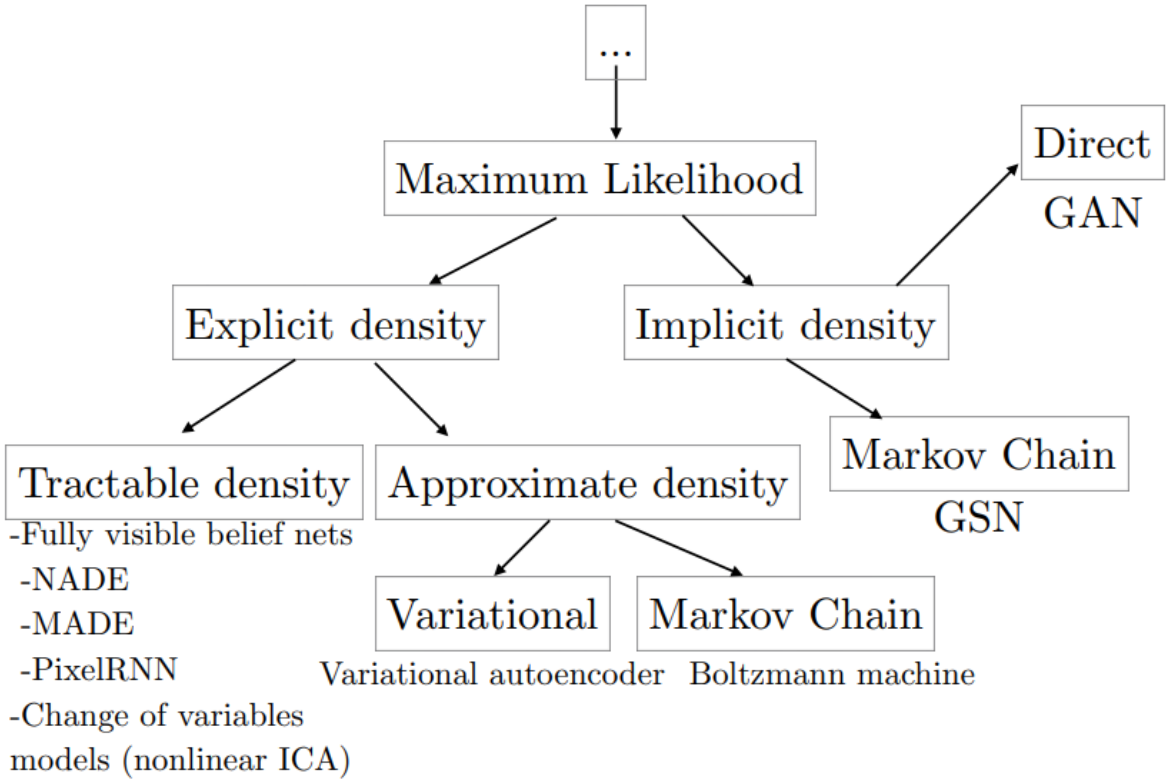
Jiarui Xu and Hao-Wen Dong

Outlines

- **Deep generative models**
  - Different generative models
  - GAN vs VAE vs Flow-based models
- **Linear algebra basics**
  - Jacobian matrix and determinant
  - Change of variable theorem
- **Normalizing Flows**
  - NICE, RealNVP and Glow
- **Autoregressive Flows**
  - MAF and IAF

Deep Generative Models

Different generative models



Ian Goodfellow, "Generative Adversarial Networks," *NeurIPS tutorial*, 2016.

Generative Adversarial Networks (GANs)

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative Adversarial Nets," *NeurIPS*, 2014.

Generative Adversarial Networks (GANs)

- A discriminator D estimates the probability of a given sample coming from the real dataset.
- A generator G outputs synthetic samples given a noise variable input.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative Adversarial Nets," *NeurIPS*, 2014.

Generative Adversarial Networks (GANs)

Define:

Generator G with parameter θ_g, Discriminator D with parameter θ_d.

Data distribution over noise input z: $p_z(z)$ (usually uniform distribution)

Data distribution over real sample: $p_{\text{data}}(x)$

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative Adversarial Nets," *NeurIPS*, 2014.

Generative Adversarial Networks (GANs)

Define:

Generator G with parameter θ_g, Discriminator D with parameter θ_d.

Data distribution over noise input z: $p_z(z)$ (usually uniform distribution)

Data distribution over real sample: $p_{\text{data}}(x)$

D should distinguish between real and fake data:

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

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative Adversarial Nets," *NeurIPS*, 2014.

Generative Adversarial Networks (GANs)

Define:

Generator G with parameter θ_g, Discriminator D with parameter θ_d.

Data distribution over noise input z: $p_z(z)$ (usually uniform distribution)

Data distribution over real sample: $p_{\text{data}}(x)$

D should distinguish between real and fake data:

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

G should be able to fool discriminator:

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log(1 - D(G(z)))$$

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative Adversarial Nets," *NeurIPS*, 2014.

Generative Adversarial Networks (GANs)

Define:

Generator G with parameter θ_g, Discriminator D with parameter θ_d.

Data distribution over noise input z: $p_z(z)$ (usually uniform distribution)

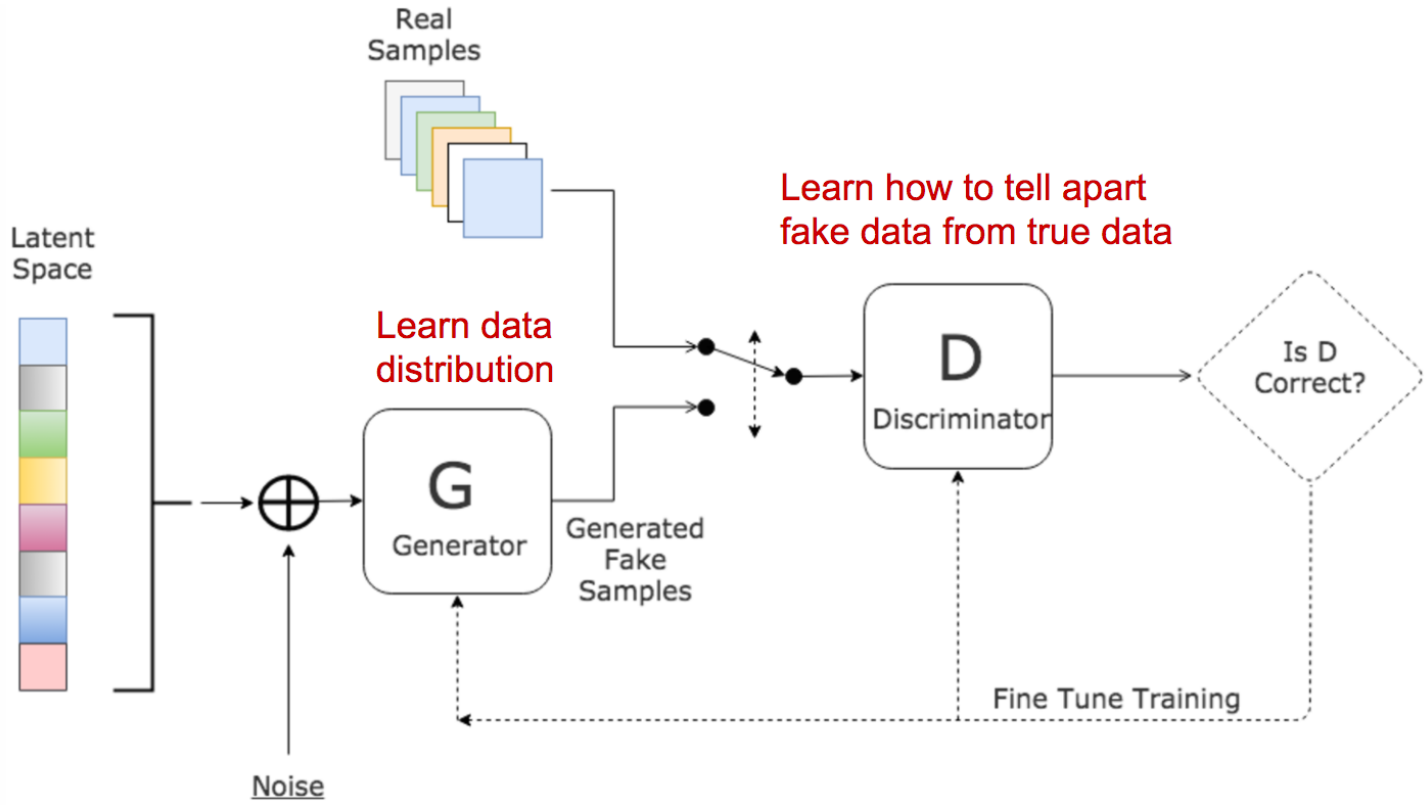
Data distribution over real sample: $p_{\text{data}}(x)$

When combining two targets together, G and D are playing a **minimax game**:

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

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, "Generative Adversarial Nets," *NeurIPS*, 2014.

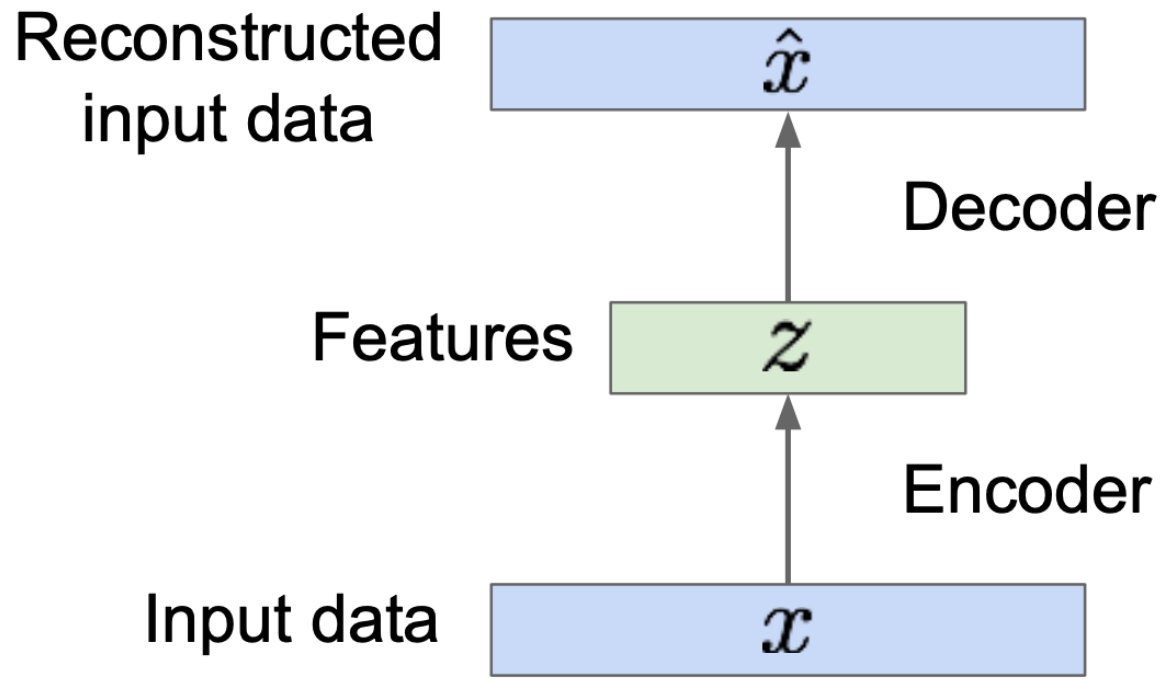
Generative Adversarial Networks (GANs)



AI Gharakhanian, "Generative Adversarial Networks," *blog post*, 2017.

Variational Autoencoders (VAEs)

Some background: Autoencoders

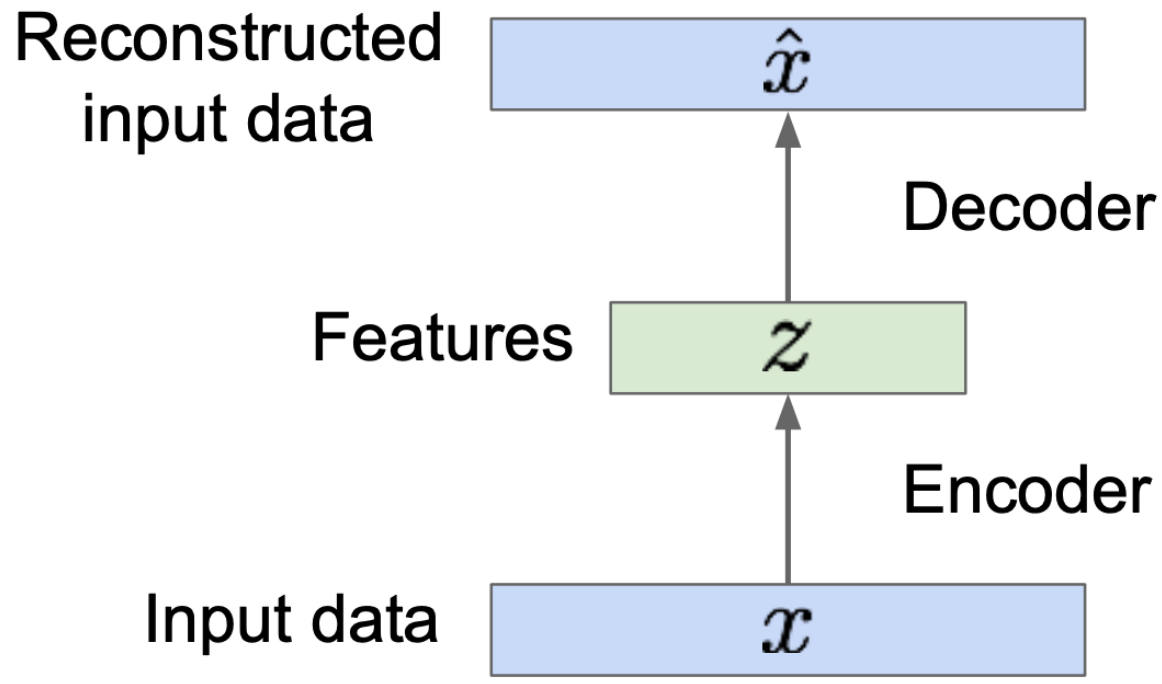


Fei-Fei Li, Ranjay Krishna, and Danfei Xu, "Lecture 11: Generative Models," *lecture note*, Stanford CS231n, 2020.

Variational Autoencoders (VAEs)

Some background: Autoencoders

- Features capture factors of variation in training data.

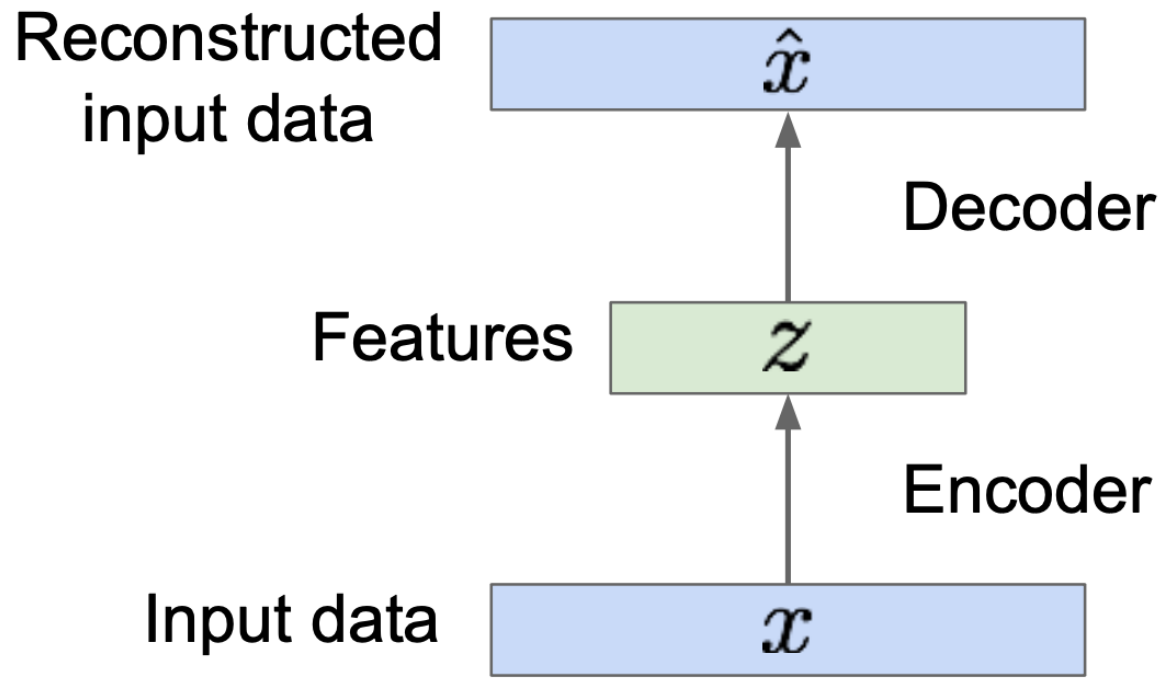


Fei-Fei Li, Ranjay Krishna, and Danfei Xu, "Lecture 11: Generative Models," *lecture note*, Stanford CS231n, 2020.

Variational Autoencoders (VAEs)

Some background: Autoencoders

- Features capture factors of variation in training data.
- But we can’t generate new images from an autoencoder because we don’t know the space of z.

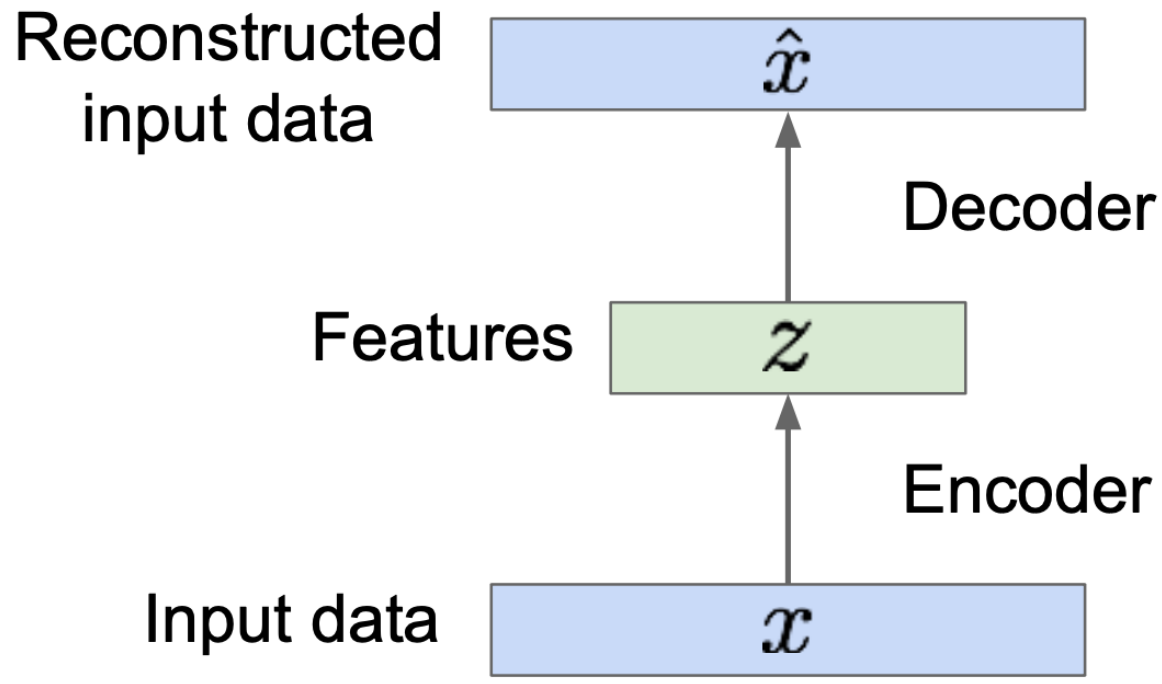


Fei-Fei Li, Ranjay Krishna, and Danfei Xu, "Lecture 11: Generative Models," *lecture note*, Stanford CS231n, 2020.

Variational Autoencoders (VAEs)

Some background: Autoencoders

- Features capture factors of variation in training data.
- But we can’t generate new images from an autoencoder because we don’t know the space of z.
- How do we make autoencoder a generative model?



Fei-Fei Li, Ranjay Krishna, and Danfei Xu, "Lecture 11: Generative Models," *lecture note*, Stanford CS231n, 2020.

Variational Autoencoders (VAEs)

We sample a z from a prior distribution $p_\theta(z)$. Then x is generated from a conditional distribution $p_\theta(x \mid z)$. The process is

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$$

Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," *ICLR*, 2014.

Variational Autoencoders (VAEs)

We sample a z from a prior distribution $p_\theta(z)$. Then x is generated from a conditional distribution $p_\theta(x \mid z)$. The process is

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$$

However, it is very expensive to check all z for integral (intractable). To narrow down the value space, consider the posterior $p_\theta(z \mid x)$ and approximate it by $q_\phi(z \mid x)$.

Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," *ICLR*, 2014.

Variational Autoencoders (VAEs)

$$
\begin{aligned}
&\log p_\theta(x) \\
&\quad = \mathbf{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta\left(x\right)\right] \\
&\quad = \mathbf{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p_\theta\left(x \mid z\right) p_\theta(z)}{p_\theta\left(z \mid x\right)}\right] \\
&\quad = \mathbf{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p_\theta\left(x \mid z\right) p_\theta(z)}{p_\theta\left(z \mid x\right)} \frac{q_\phi\left(z \mid x\right)}{q_\phi\left(z \mid x\right)}\right] \\
&\quad = \mathbf{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta\left(x \mid z\right)\right] - \mathbf{E}_{z\sim q_\phi(z|x)}\left[\log \frac{q_\phi\left(z \mid x\right)}{p_\theta(z)}\right] + \mathbf{E}_{z\sim q_\phi(z|x)}\left[\log \frac{q_\phi\left(z \mid x\right)}{p_\theta\left(z \mid x\right)}\right] \\
&\quad = \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta\left(x \mid z\right)\right] - D_{KL}\left(q_\phi\left(z \mid x\right) \| p_\theta(z)\right) + D_{KL}\left(q_\phi\left(z \mid x\right) \| p_\theta\left(z \mid x\right)\right) \\
&\quad \geq \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p_\theta\left(x \mid z\right)\right] - D_{KL}\left(q_\phi\left(z \mid x\right) \| p_\theta(z)\right) \\
&\quad = \mathrm{ELBO}(x; \theta, \phi)
\end{aligned}
$$

Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," *ICLR*, 2014.

Variational Autoencoders (VAEs)

We sample a z from a prior distribution $p_\theta(z)$. Then x is generated from a conditional distribution $p_\theta(x \mid z)$. The process is

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$$

However, it is very expensive to check all z for integral (intractable). To narrow down the value space, consider the posterior $p_\theta(z \mid x)$ and approximate it by $q_\phi(z \mid x)$.

Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," *ICLR*, 2014.

Variational Autoencoders (VAEs)

We sample a z from a prior distribution $p_\theta(z)$. Then x is generated from a conditional distribution $p_\theta(x \mid z)$. The process is

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$$

However, it is very expensive to check all z for integral (intractable). To narrow down the value space, consider the posterior $p_\theta(z \mid x)$ and approximate it by $q_\phi(z \mid x)$.

The data likelihood

$$\begin{aligned}\log p_\theta(x) &= \mathbb{E}_{z \sim q_\phi(z|x)} \left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z)\right) + D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z \mid x)\right) \\ &\geq \mathbb{E}_{z \sim q_\phi(z|x)} \left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z)\right) = \mathrm{ELBO}(x; \theta, \phi)\end{aligned}$$

Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," *ICLR*, 2014.

Variational Autoencoders (VAEs)

We sample a z from a prior distribution $p_\theta(z)$. Then x is generated from a conditional distribution $p_\theta(x \mid z)$. The process is

$$p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x} \mid \mathbf{z}) p_\theta(\mathbf{z}) d\mathbf{z}$$

However, it is very expensive to check all z for integral (intractable). To narrow down the value space, consider the posterior $p_\theta(z \mid x)$ and approximate it by $q_\phi(z \mid x)$.

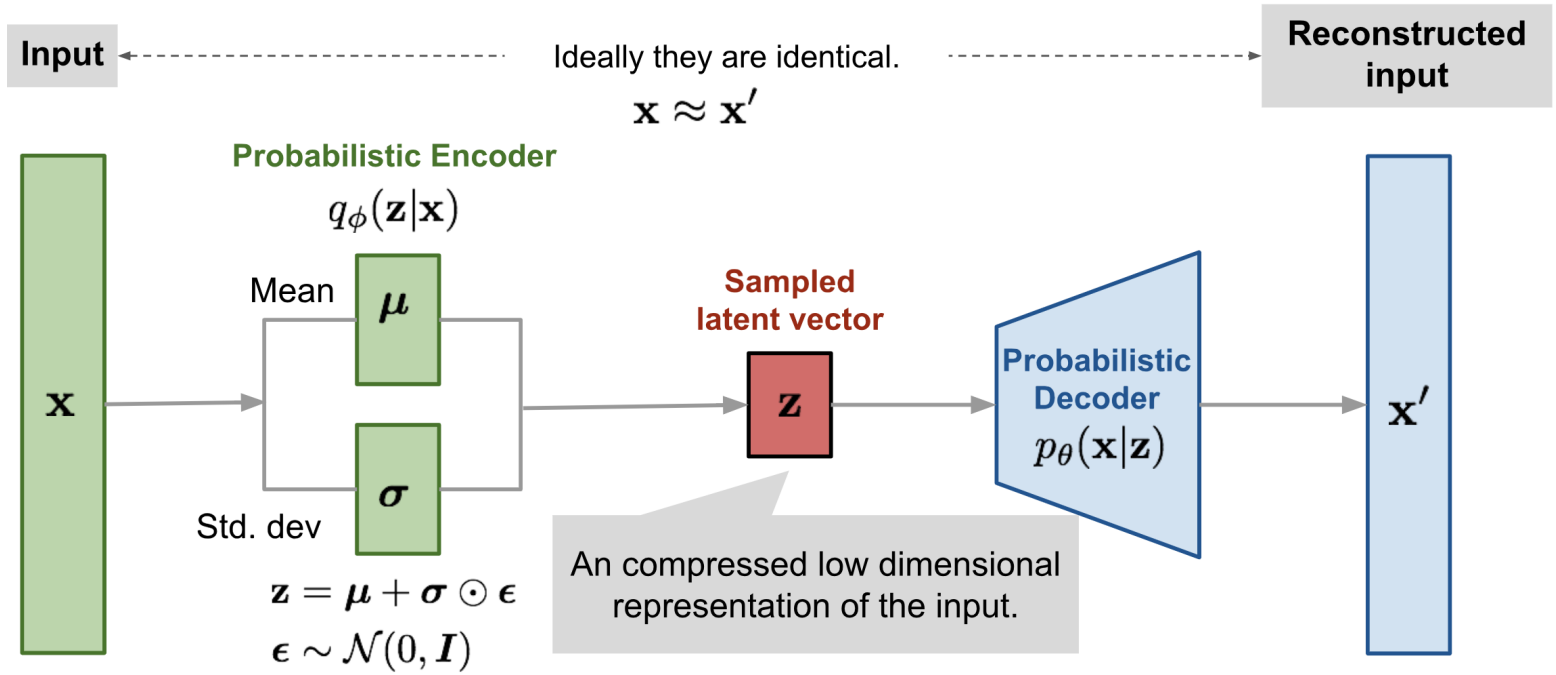
The data likelihood

$$\begin{aligned}\log p_\theta(x) &= \mathbb{E}_{z \sim q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z)\right) + D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z \mid x)\right) \\ &\geq \mathbb{E}_{z \sim q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] - D_{KL}\left(q_\phi(z \mid x) \| p_\theta(z)\right) = \mathrm{ELBO}(x; \theta, \phi)\end{aligned}$$

$\mathrm{ELBO}(x; \theta, \phi)$ is tractable, $\max_\theta \log p_\theta(x) \rightarrow \max_{\theta, \phi} \mathrm{ELBO}(x; \theta, \phi)$.

Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," *ICLR*, 2014.

Variational Autoencoders (VAEs)



Lilian Weng, "From Autoencoder to Beta-VAE," *blog post*, 2018.

GANs vs VAEs vs Flow-based models

Optimization target

GANs vs VAEs vs Flow-based models

Optimization target

GAN:

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

GANs vs VAEs vs Flow-based models

Optimization target

GAN:

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

VAE:

$$\max_{\theta, \phi} \mathbb{E}_{z \sim q_\phi(z|x)} \left[\log p_\theta \left(x \mid z \right) \right] - D_{KL} \left(q_\phi \left(z \mid x \right) \| p_\theta(z) \right)$$

$$= \text{ELBO}(x; \theta, \phi)$$

GANs vs VAEs vs Flow-based models

Optimization target

GAN:

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

VAE:

$$\max_{\theta, \phi} \mathbb{E}_{z \sim q_\phi(z|x)} \left[\log p_\theta \left(x \mid z \right) \right] - D_{KL} \left(q_\phi \left(z \mid x \right) \| p_\theta(z) \right)$$

$$= \text{ELBO}(x; \theta, \phi)$$

Flow-based generative models:

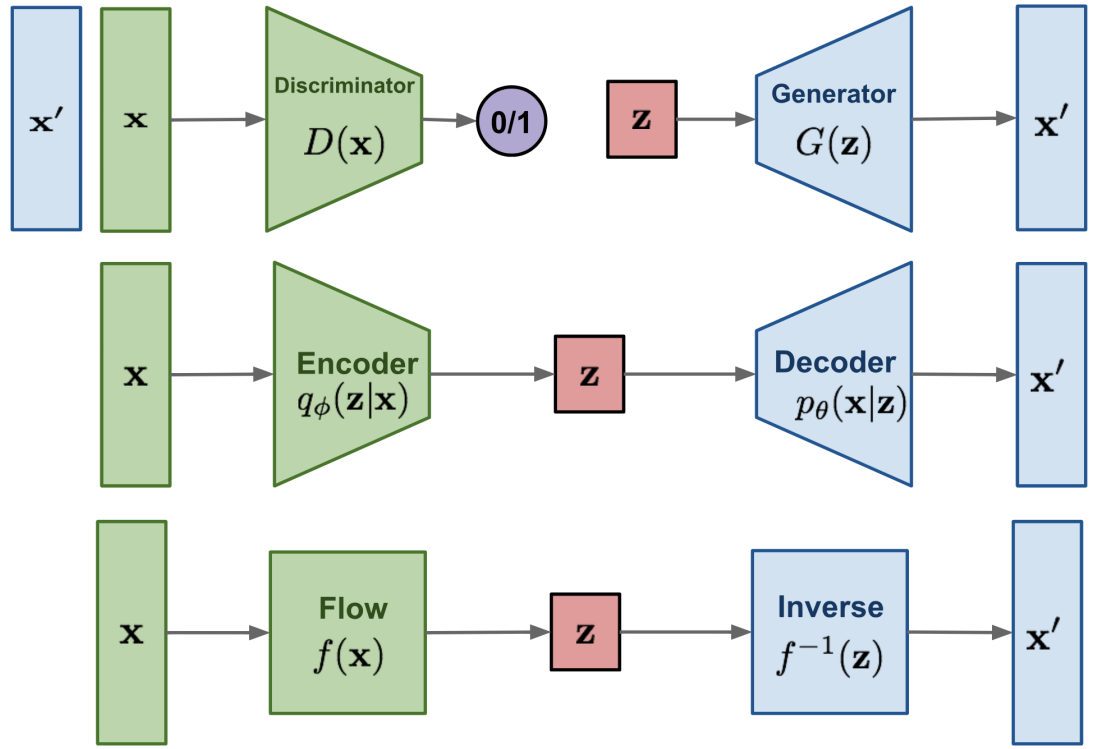
$$\max_{\theta} \mathbb{E}_{x \sim p_{\text{data}}} \log p_\theta(x)$$

GANs vs VAEs vs Flow-based models

GAN: minimax the classification error loss.

VAE: maximize ELBO.

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



Lilian Weng, "Flow-based Deep Generative Models," *blog post*, 2018.

How to estimate data likelihood directly?

Linear Algebra Basics

Jacobian matrix

Given a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ that takes as input a n-dimensional input vector $\mathbf{x}$ and output a m-dimensional vector, the Jacobian matrix of $\mathbf{f}$ is defined as

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

which is the matrix of all first-order partial derivatives. The entry on the i-th row and j-th column is

$$\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$$

Change of variable theorem

Given some random variable $z \sim \pi(z)$ and a invertible mapping $x = f(z)$ (i.e., $z = f^{-1}(x) = g(x)$). Then, the distribution of x is

$$p(x) = \pi(z) \left| \frac{dz}{dx} \right| = \pi(g(x)) \left| \frac{dg}{dx} \right|$$

Change of variable theorem

Given some random variable $z \sim \pi(z)$ and a invertible mapping $x = f(z)$ (i.e., $z = f^{-1}(x) = g(x)$). Then, the distribution of x is

$$p(x) = \pi(z) \left| \frac{dz}{dx} \right| = \pi(g(x)) \left| \frac{dg}{dx} \right|$$

The multivariate version takes the following form:

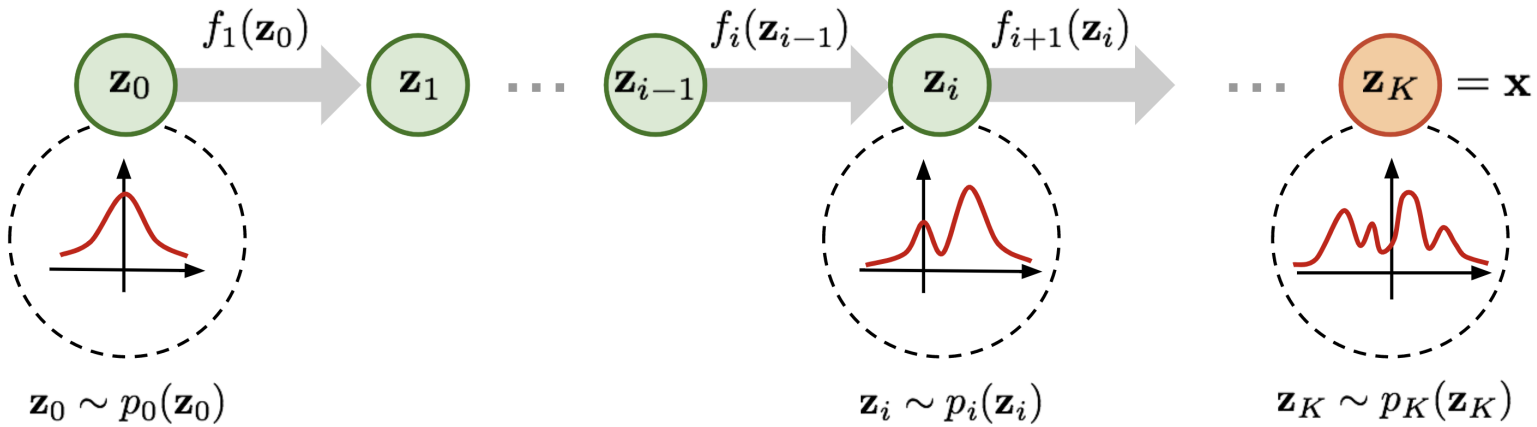
$$p(\mathbf{x}) = \pi(\mathbf{z}) \left| \det \frac{d\mathbf{z}}{d\mathbf{x}} \right| = \pi(g(\mathbf{x})) \left| \det \frac{dg}{d\mathbf{x}} \right|$$

where $\det \frac{dg}{d\mathbf{x}}$ is the *Jacobian determinant* of g.

Normalizing Flows

Normalizing flows

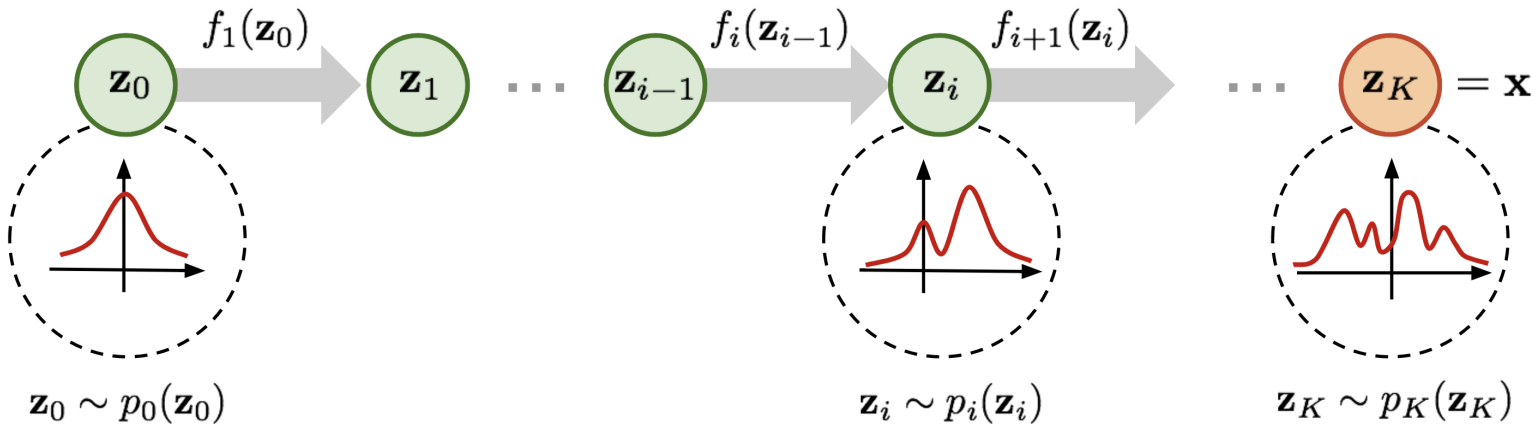
Key: Transform a simple distribution into a complex one by applying a sequence of *invertible transformations*.



Lilian Weng, "Flow-based Deep Generative Models," *blog post*, 2018.

Normalizing flows

Key: Transform a simple distribution into a complex one by applying a sequence of *invertible transformations*.



- In each step, substitute the variable with the new one by change of variables theorem.
- Eventually, obtain a distribution close enough to the target distribution.

Lilian Weng, "Flow-based Deep Generative Models," *blog post*, 2018.

Normalizing flows

For each step, we have $\mathbf{z}_i \sim p_i(\mathbf{z}_i)$, $\mathbf{z}_i = f_i(\mathbf{z}_{i-1})$ and $\mathbf{z}_{i-1} = g_i(\mathbf{z}_i)$. Now,

$$
\begin{aligned}
p_i(\mathbf{z}_i) &= p_{i-1}(g_i(\mathbf{z}_i)) \left| \det \frac{dg_i(\mathbf{z}_i)}{d\mathbf{z}_i} \right| && \text{(by change of variables theorem)} \\
&= p_{i-1}(\mathbf{z}_{i-1}) \left| \det \frac{d\mathbf{z}_{i-1}}{df_i(\mathbf{z}_{i-1})} \right| && \text{(by definition)} \\
&= p_{i-1}(\mathbf{z}_{i-1}) \left| \det \left(\frac{df_i(\mathbf{z}_{i-1})}{d\mathbf{z}_{i-1}} \right)^{-1} \right| && \text{(by inverse function theorem)} \\
&= p_{i-1}(\mathbf{z}_{i-1}) \left| \det \frac{df_i}{d\mathbf{z}_{i-1}} \right|^{-1} && \text{(by } \det M \det(M^{-1}) = \det I = 1\text{)}
\end{aligned}
$$

Thus, we have $\log p_i(\mathbf{z}_i) = \log p_{i-1}(\mathbf{z}_{i-1}) - \log \left| \det \frac{df_i}{d\mathbf{z}_{i-1}} \right|$.

Danilo Jimenez Rezende and Shakir Mohamed, "Variational Inference with Normalizing Flows," *ICML*, 2015.

Normalizing flows

Now, we obtain $\log p_i(\mathbf{z}_i) = \log p_{i-1}(\mathbf{z}_{i-1}) - \log \left| \det \frac{df_i}{d\mathbf{z}_{i-1}} \right|$

Recall that $\mathbf{x} = \mathbf{z}_K = f_K \circ f_{K-1} \circ \cdots \circ f_1(\mathbf{z_0})$.

Thus, we have

$$\begin{aligned}
\log p(\mathbf{x}) &= \log p_K(\mathbf{z}_K) \\
&= \log p_{K-1}(\mathbf{z}_{K-1}) - \log \left| \det \frac{df_K}{d\mathbf{z}_{K-1}} \right| \\
&= \dots \\
&= \log p_0(\mathbf{z}_0) - \sum_{i=1}^{K} \log \left| \det \frac{df_i}{d\mathbf{z}_{i-1}} \right|
\end{aligned}$$

Danilo Jimenez Rezende and Shakir Mohamed, "Variational Inference with Normalizing Flows," *ICML*, 2015.

Normalizing flows

In normalizing flows, the exact log-likelihood $\log p(\mathbf{x})$ of input data x is

$$\log p(\mathbf{x}) = \log p_0(\mathbf{z}_0) - \sum_{i=1}^{K} \log \left| \det \frac{df_i}{d\mathbf{z}_{i-1}} \right|$$

Danilo Jimenez Rezende and Shakir Mohamed, "Variational Inference with Normalizing Flows," *ICML*, 2015.

Normalizing flows

In normalizing flows, the exact log-likelihood $\log p(\mathbf{x})$ of input data x is

$$\log p(\mathbf{x}) = \log p_0(\mathbf{z}_0) - \sum_{i=1}^{K} \log \left| \det \frac{df_i}{d\mathbf{z}_{i-1}} \right|$$

To make the computation tractable, it requires

- f_i is easily invertible
- The Jacobian determinant of f_i is easy to compute

Danilo Jimenez Rezende and Shakir Mohamed, "Variational Inference with Normalizing Flows," *ICML*, 2015.

Normalizing flows

In normalizing flows, the exact log-likelihood $\log p(\mathbf{x})$ of input data x is

$$\log p(\mathbf{x}) = \log p_0(\mathbf{z}_0) - \sum_{i=1}^{K} \log \left| \det \frac{df_i}{d\mathbf{z}_{i-1}} \right|$$

To make the computation tractable, it requires

- f_i is easily invertible
- The Jacobian determinant of f_i is easy to compute

Then, we can train the model by maximizing the log-likelihood over some training dataset $\mathcal{D}$

$$LL(\mathcal{D}) = \sum_{\mathbf{x} \in \mathcal{D}} \log p(\mathbf{x})$$

Danilo Jimenez Rezende and Shakir Mohamed, "Variational Inference with Normalizing Flows," *ICML*, 2015.

NICE

The core idea behind NICE (Non-linear Independent Components Estimation) is to

1. split $\mathbf{x} \in \mathbb{R}^D$ into two blocks $\mathbf{x}_1 \in \mathbb{R}^d$ and $\mathbf{x}_2 \in \mathbb{R}^{D-d}$

Laurent Dinh, David Krueger, and Yoshua Bengio, "NICE: Non-linear Independent Components Estimation," *ICLR*, 2015.

NICE

The core idea behind NICE (Non-linear Independent Components Estimation) is to

1. split $\mathbf{x} \in \mathbb{R}^D$ into two blocks $\mathbf{x}_1 \in \mathbb{R}^d$ and $\mathbf{x}_2 \in \mathbb{R}^{D-d}$
2. apply the following transformation from $(\mathbf{x}_1, \mathbf{x}_2)$ to $(\mathbf{y}_1, \mathbf{y}_2)$

$$\begin{cases} \mathbf{y}_1 &= \mathbf{x}_1 \\ \mathbf{y}_2 &= \mathbf{x}_2 + m(\mathbf{x}_1) \end{cases}$$

where $m(\cdot)$ is an arbitrarily function (e.g., a deep neural network).

Laurent Dinh, David Krueger, and Yoshua Bengio, "NICE: Non-linear Independent Components Estimation," *ICLR*, 2015.

NICE - Additive coupling layers

The transformation

$$\begin{cases} \mathbf{y}_1 &= \mathbf{x}_1 \\ \mathbf{y}_2 &= \mathbf{x}_2 + m(\mathbf{x}_1) \end{cases}$$

- is trivially invertible.

$$\begin{cases} \mathbf{x}_1 &= \mathbf{y}_1 \\ \mathbf{x}_2 &= \mathbf{y}_2 - m(\mathbf{y}_1) \end{cases}$$

Laurent Dinh, David Krueger, and Yoshua Bengio, "NICE: Non-linear Independent Components Estimation," *ICLR*, 2015.

NICE - Additive coupling layers

The transformation

$$\begin{cases} \mathbf{y}_1 &= \mathbf{x}_1 \\ \mathbf{y}_2 &= \mathbf{x}_2 + m(\mathbf{x}_1) \end{cases}$$

- has a unit Jacobian determinant.

$$\mathbf{J} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0}_{d \times (D-d)} \\ \frac{\partial m(\mathbf{x}_1)}{\partial \mathbf{x}_1} & \mathbf{I}_{D-d} \end{bmatrix}$$

$$\det(\mathbf{J}) = \mathbf{I}$$

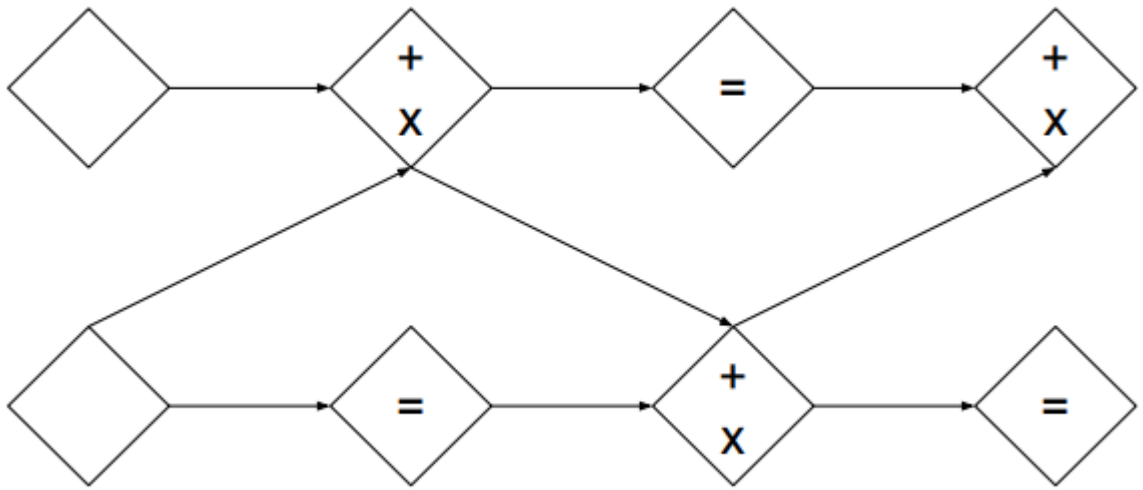
Note that NICE is a type of *volume-preserving flows* as it has a unit Jacobian determinant.

Laurent Dinh, David Krueger, and Yoshua Bengio, "NICE: Non-linear Independent Components Estimation," *ICLR*, 2015.

NICE - Alternating pattern

Some dimensions remain unchanged after the transform

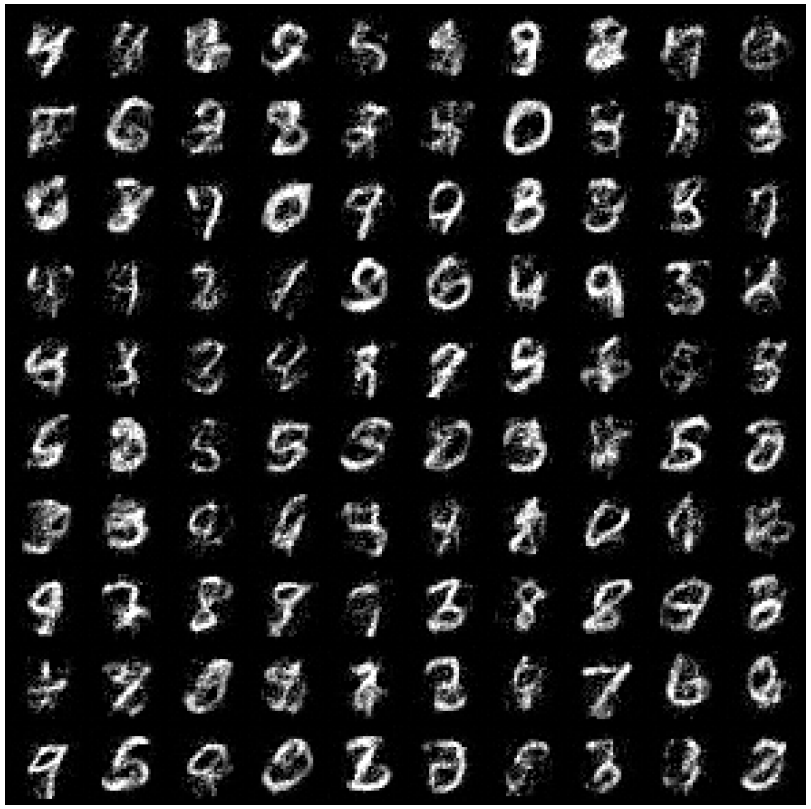
- alternate the dimensions being modified
- 3 coupling layers are necessary to allow all dimensions to influence one another



Laurent Dinh, David Krueger, and Yoshua Bengio, "NICE: Non-linear Independent Components Estimation," *ICLR*, 2015.
Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio, "Density Estimation using Real NVP," *ICLR*, 2017.

NICE - Experiments on MNIST

Settings: 784 dimensions (28×28), 6 additive coupling layers



RealNVP

The core idea behind RealNVP (Real-valued Non-Volume Preserving) is to

1. split $\mathbf{x} \in \mathbb{R}^D$ into two blocks $\mathbf{x}_1 \in \mathbb{R}^d$ and $\mathbf{x}_2 \in \mathbb{R}^{D-d}$

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio, "Density Estimation using Real NVP," *ICLR*, 2017.

RealNVP

The core idea behind RealNVP (Real-valued Non-Volume Preserving) is to

1. split $\mathbf{x} \in \mathbb{R}^D$ into two blocks $\mathbf{x}_1 \in \mathbb{R}^d$ and $\mathbf{x}_2 \in \mathbb{R}^{D-d}$
2. apply the following transformation from $(\mathbf{x}_1, \mathbf{x}_2)$ to $(\mathbf{y}_1, \mathbf{y}_2)$

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

where $s(\cdot)$ and $t(\cdot)$ are *scale* and *translation* functions that map $\mathbb{R}^d$ to $\mathbb{R}^{D-d}$, and $\odot$ denotes the element-wise product.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio, "Density Estimation using Real NVP," *ICLR*, 2017.

RealNVP

The core idea behind RealNVP (Real-valued Non-Volume Preserving) is to

1. split $\mathbf{x} \in \mathbb{R}^D$ into two blocks $\mathbf{x}_1 \in \mathbb{R}^d$ and $\mathbf{x}_2 \in \mathbb{R}^{D-d}$
2. apply the following transformation from $(\mathbf{x}_1, \mathbf{x}_2)$ to $(\mathbf{y}_1, \mathbf{y}_2)$

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

where $s(\cdot)$ and $t(\cdot)$ are *scale* and *translation* functions that map $\mathbb{R}^d$ to $\mathbb{R}^{D-d}$, and $\odot$ denotes the element-wise product.

(Note that NICE does not have the scaling term.)

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio, "Density Estimation using Real NVP," *ICLR*, 2017.

RealNVP - Affine coupling layers

The transformation

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

- is easily invertible.

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

  (Note that it does not involve computing s^{-1} and t^{-1}.)

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio, "Density Estimation using Real NVP," *ICLR*, 2017.

RealNVP - Affine coupling layers

The transformation

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

- has a Jacobian determinant that is easy to compute.

$$\mathbf{J} = \begin{bmatrix} \mathbf{I}_d & \mathbf{0}_{d \times (D-d)} \\ \frac{\partial \mathbf{y}_{d+1:D}}{\partial \mathbf{x}_{1:d}} & \operatorname{diag}\left(e^{s(\mathbf{x}_{1:d})}\right) \end{bmatrix}$$

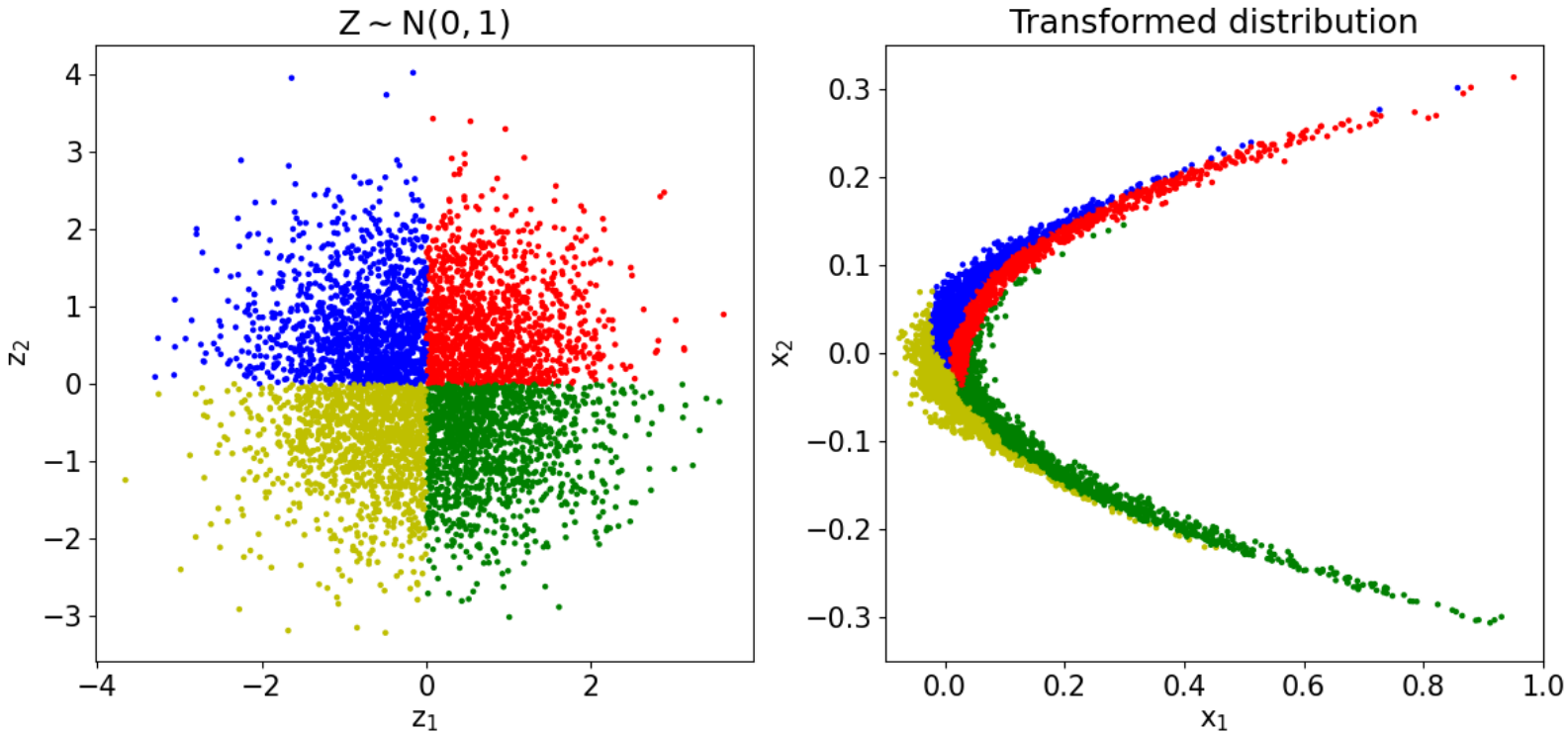
$$\det(\mathbf{J}) = \prod_{j=1}^{D-d} e^{s(\mathbf{x}_{1:d})_j} = \exp\left(\sum_{j=1}^{D-d} s(\mathbf{x}_{1:d})_j\right)$$

(Note that it does not involve computing the Jacobian of s and t.)

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio, "Density Estimation using Real NVP," *ICLR*, 2017.

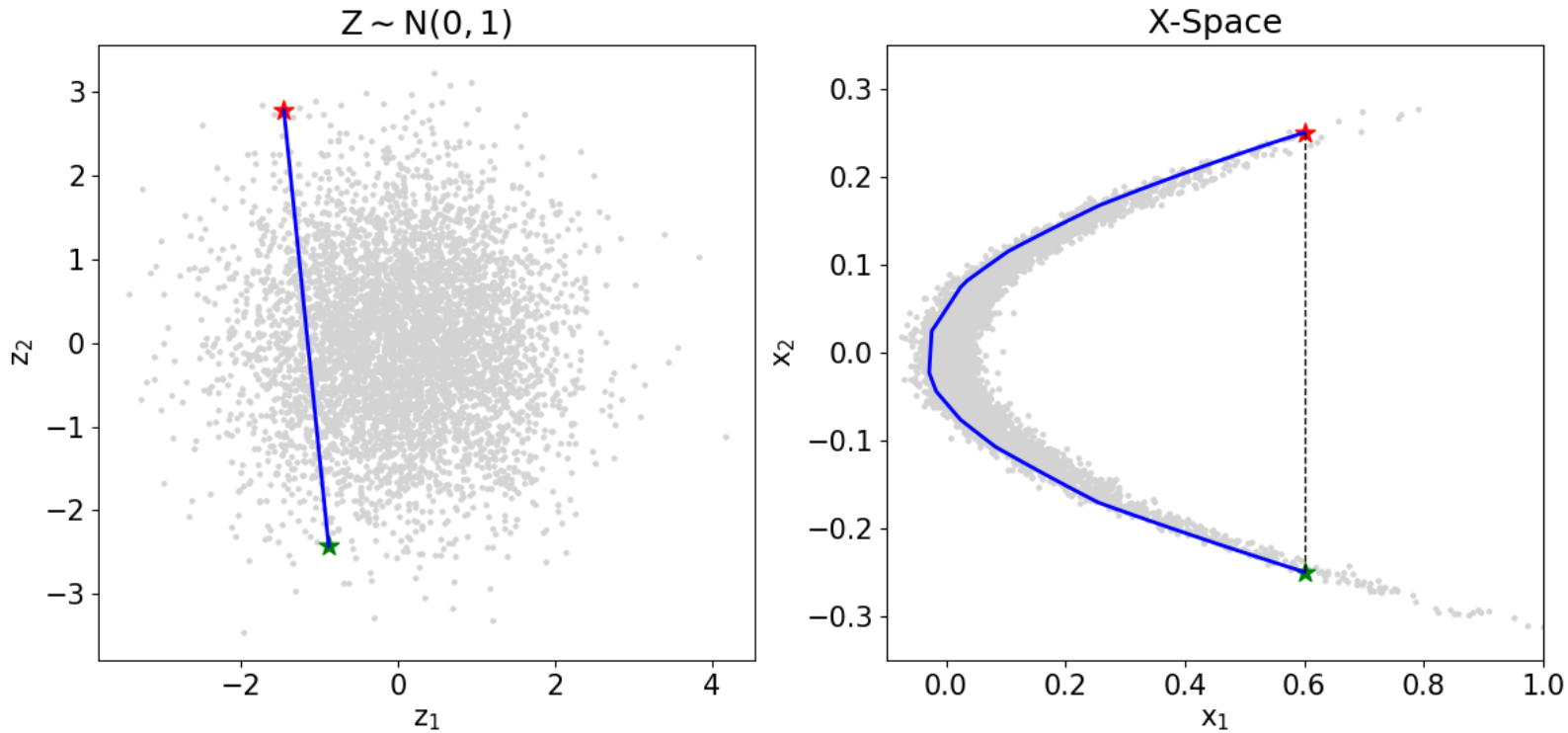
RealNVP - Experiments on toy data

Settings: 2D data, 5 affine coupling layers



RealNVP - Experiments on toy data

Settings: 2D data, 5 affine coupling layers

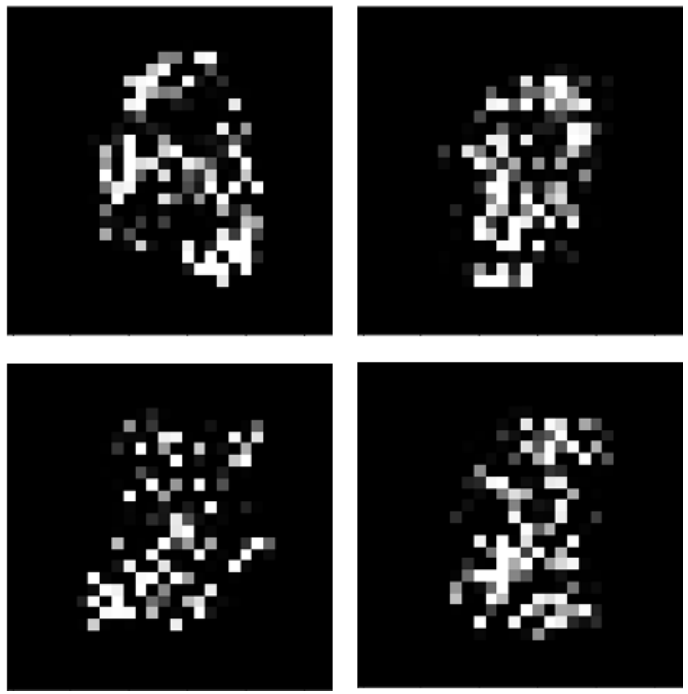


RealNVP - Experiments on MNIST

Settings: 784 dimensions (28×28), 5 affine coupling layers

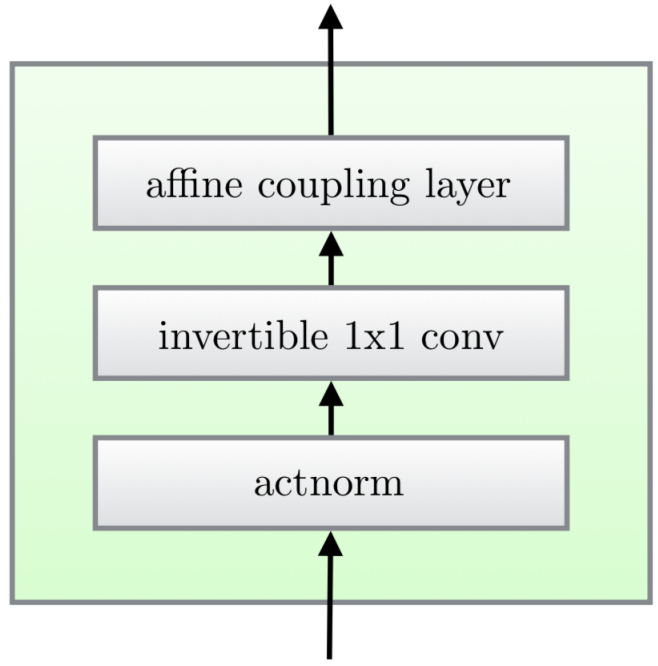
Digit 2

All digits



Glow

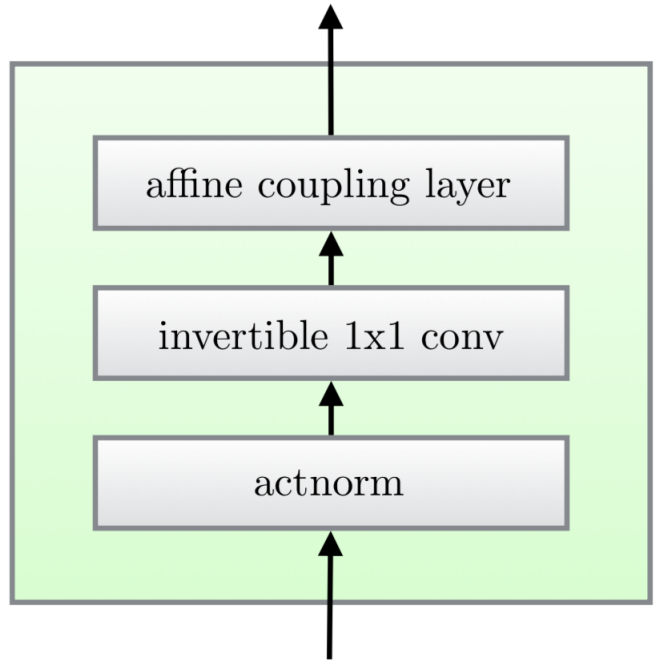
- Actnorm:
  - Forward: $\mathbf{y} = \mathbf{s} \odot \mathbf{x} + \mathbf{b}$
  - Backward: $\mathbf{x} = \mathbf{s} \odot (\mathbf{y} - \mathbf{b})$
  - Log-determinant: $h \cdot w \cdot \sum_i \log |\mathbf{s}_i|$



Diederik P. Kingma and Prafulla Dhariwal, "Glow: Generative Flow with Invertible 1×1 Convolutions," *NeurIPS*, 2018

Glow

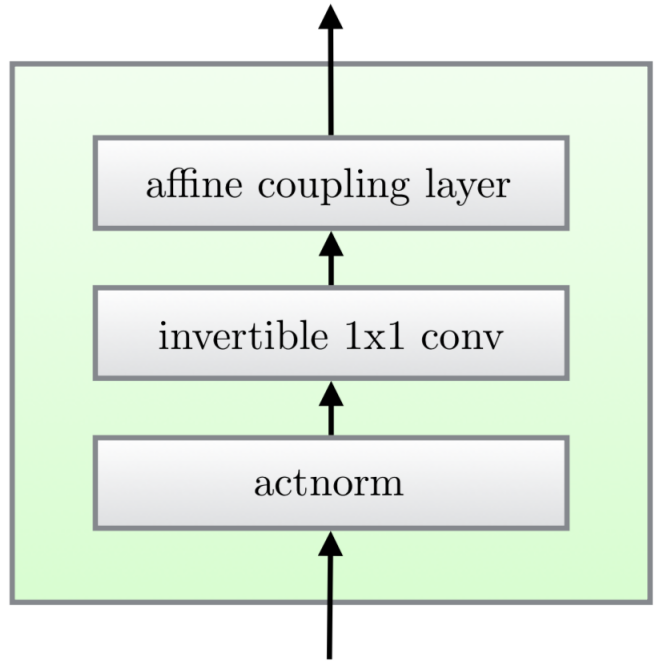
- Actnorm:
  - Forward: $\mathbf{y} = \mathbf{s} \odot \mathbf{x} + \mathbf{b}$
  - Backward: $\mathbf{x} = \mathbf{s} \odot (\mathbf{y} - \mathbf{b})$
  - Log-determinant: $h \cdot w \cdot \sum_i \log |\mathbf{s}_i|$
- Invertible 1×1 convolution:
  - Forward: $\mathbf{y} = \mathbf{W}\mathbf{x}$
  - Backward: $\mathbf{x} = \mathbf{W}^{-1}\mathbf{y}$
  - Log-determinant: $h \cdot w \cdot \log |\det \mathbf{W}|$



Diederik P. Kingma and Prafulla Dhariwal, "Glow: Generative Flow with Invertible 1×1 Convolutions," *NeurIPS*, 2018

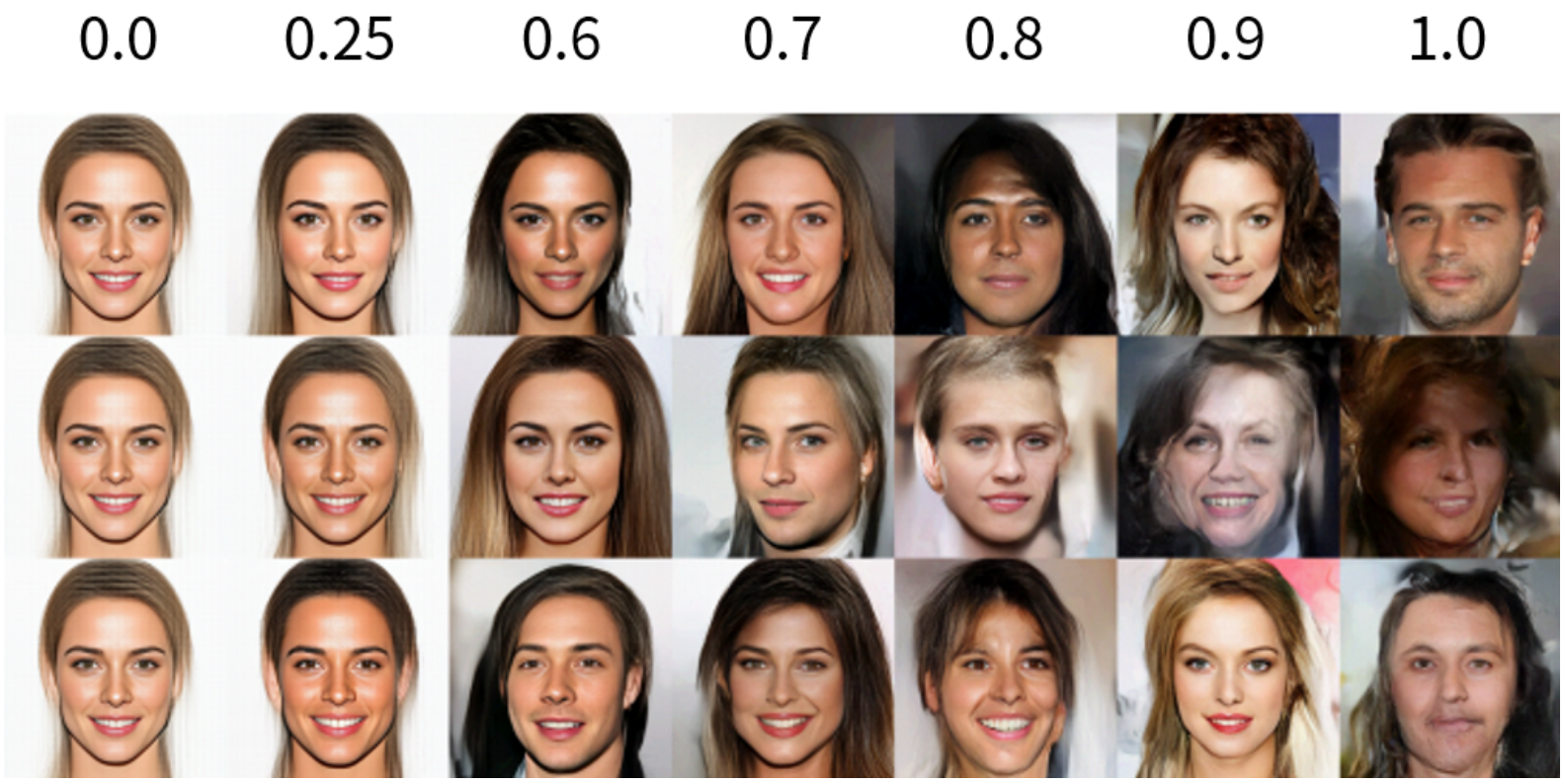
Glow

- Actnorm:
  - Forward: $\mathbf{y} = \mathbf{s} \odot \mathbf{x} + \mathbf{b}$
  - Backward: $\mathbf{x} = \mathbf{s} \odot (\mathbf{y} - \mathbf{b})$
  - Log-determinant: $h \cdot w \cdot \sum_i \log |\mathbf{s}_i|$
- Invertible 1×1 convolution:
  - Forward: $\mathbf{y} = \mathbf{W}\mathbf{x}$
  - Backward: $\mathbf{x} = \mathbf{W}^{-1}\mathbf{y}$
  - Log-determinant: $h \cdot w \cdot \log |\det \mathbf{W}|$
- Affine coupling Layer: same as RealNVP



Diederik P. Kingma and Prafulla Dhariwal, "Glow: Generative Flow with Invertible 1×1 Convolutions," *NeurIPS*, 2018

Glow - Samples



Diederik P. Kingma and Prafulla Dhariwal, "Glow: Generative Flow with Invertible 1×1 Convolutions," *NeurIPS*, 2018

Autoregressive Flows

Autoregressive flows

Key: Model the transformation in a normalizing flow as an *autoregressive model*.

In an autoregressive model, we assume that *the current output depends only on the data observed in the past* and factorize the joint probability $p(x_1, x_2, \ldots, x_D)$ into the product of the probability of observing x_i conditioned on the past observations $x_1, x_2, \ldots, x_{i-1}$.

$$
\begin{aligned}
p(\mathbf{x}) &= p(x_1, x_2, \ldots, x_D) \\
&= p(x_1)\, p(x_2|x_1)\, p(x_3|x_1, x_2) \ \ldots \ p(x_D|x_1, x_2, \ldots, x_{D-1}) \\
&= \prod_{i=1}^{D} p(x_i|x_1, x_2, \ldots, x_{i-1}) \\
&= \prod_{i=1}^{D} p(x_i|\mathbf{x}_{1:i-1})
\end{aligned}
$$

Lilian Weng, "Flow-based Deep Generative Models," *blog post*, 2018.

Masked autoregressive flow (MAF)

Given two random variables $\mathbf{z} \sim \pi(\mathbf{z})$ and $\mathbf{x} \sim p(\mathbf{x})$ where $\pi(\mathbf{z})$ is known but $p(\mathbf{x})$ is unknown. Masked autoregressive flow (MAF) aims to learn $p(x)$.

- Sampling:

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

  Note that this computation is slow as it is sequential and autoregressive.

George Papamakarios, Theo Pavlakou, and Iain Murray, "Masked Autoregressive Flow for Density Estimation," *NeurIPS*, 2017.
Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle, "MADE: Masked Autoencoder for Distribution Estimation," *ICML*, 2015.

Masked autoregressive flow (MAF)

Given two random variables $\mathbf{z} \sim \pi(\mathbf{z})$ and $\mathbf{x} \sim p(\mathbf{x})$ where $\pi(\mathbf{z})$ is known but $p(\mathbf{x})$ is unknown. Masked autoregressive flow (MAF) aims to learn $p(x)$.

- Sampling:

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

  Note that this computation is slow as it is sequential and autoregressive.

- Density estimation:

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

  Note that this computation can be fast if we use the *masking* approach introduced in MADE as it only requires one single pass to the network.

George Papamakarios, Theo Pavlakou, and Iain Murray, "Masked Autoregressive Flow for Density Estimation," *NeurIPS*, 2017.
Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle, "MADE: Masked Autoencoder for Distribution Estimation," *ICML*, 2015.

Inverse autoregressive flow (IAF)

In MAF, we have $x_i = z_i \odot \sigma_i(\mathbf{x}_{1:i-1}) + \mu_i(\mathbf{x}_{1:i-1})$. We can reverse it into

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

Now, if we swap $\mathbf{x}$ and $\mathbf{z}$ (let $\tilde{\mathbf{z}} = \mathbf{x}$ and $\tilde{\mathbf{x}} = \mathbf{z}$), we get the inverse autoregressive flow (IAF)

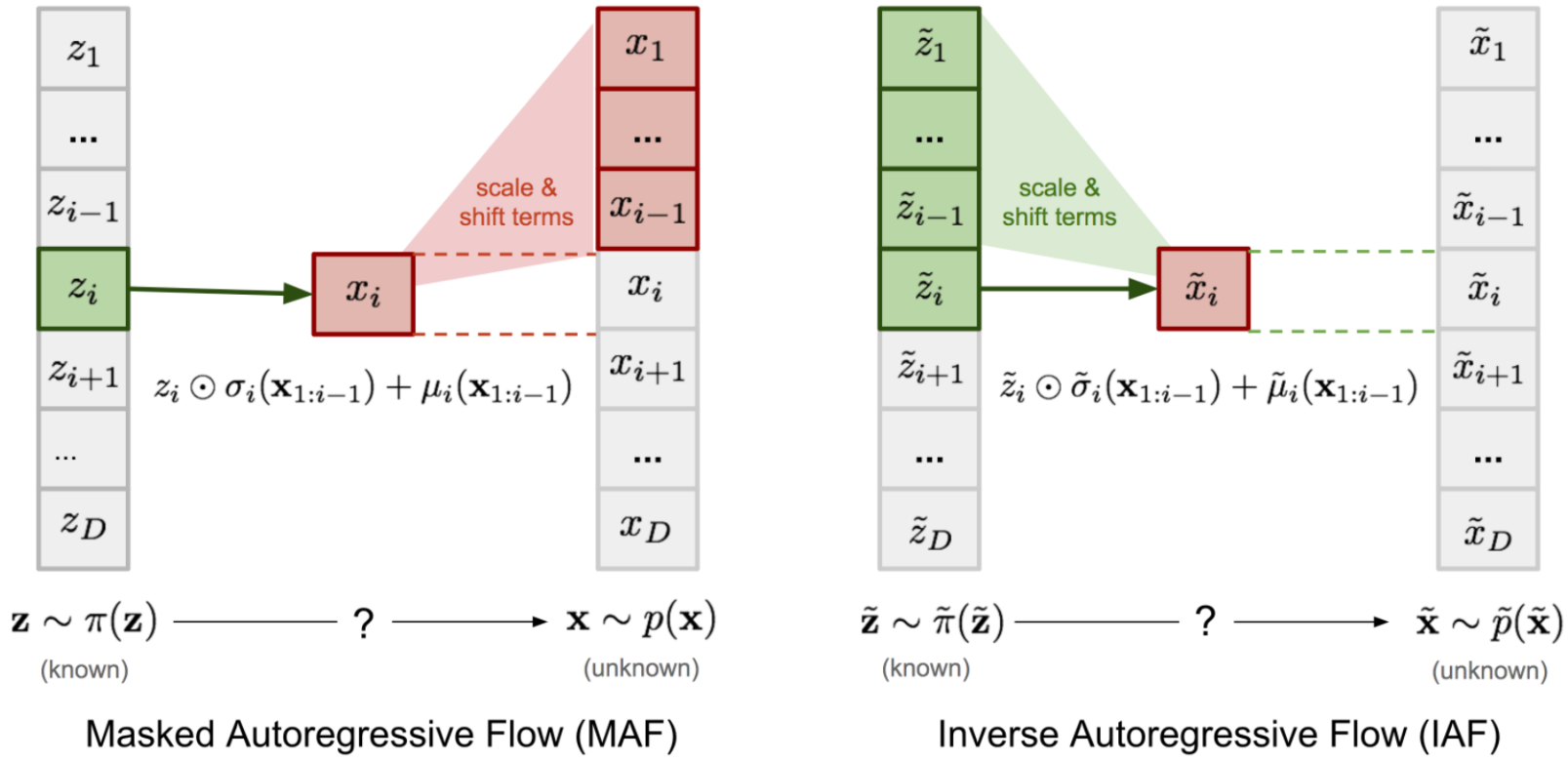
$$\begin{aligned}\tilde{x}_i &= \tilde{z}_i \odot \frac{1}{\sigma_i(\tilde{\mathbf{z}}_{1:i-1})} - \frac{\mu_i(\tilde{\mathbf{z}}_{1:i-1})}{\sigma_i(\tilde{\mathbf{z}}_{1:i-1})} \\ &= \tilde{z}_i \odot \tilde{\sigma}_i(\tilde{\mathbf{z}}_{1:i-1}) + \tilde{\mu}_i(\tilde{\mathbf{z}}_{1:i-1})\end{aligned}$$

where

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

Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling, "Improved Variational Inference with Inverse Autoregressive Flow," *NeurIPS*, 2016.

MAF vs IAF



(Note that $\tilde{\mathbf{z}} = \mathbf{x}$, $\tilde{\mathbf{x}} = \mathbf{z}$, $\tilde{\pi} = p$ and $\tilde{p} = \pi$.)

Lilian Weng, "Flow-based Deep Generative Models," *blog post*, 2018.

MAF vs IAF

	MAF	IAF
Base distribution	$\mathbf{z} \sim \pi(\mathbf{z})$	$\mathbf{x} \sim p(\mathbf{x})$
Target distribution	$\tilde{\mathbf{z}} \sim \tilde{\pi}(\tilde{\mathbf{z}})$	$\tilde{\mathbf{x}} \sim \tilde{p}(\tilde{\mathbf{x}})$
Model	$x_i = z_i \odot \sigma_i(\mathbf{x}_{1:i-1}) + \mu_i(\mathbf{x}_{1:i-1})$	$\tilde{x}_i = \tilde{z}_i \odot \tilde{\sigma}_i(\tilde{\mathbf{z}}_{1:i-1}) + \tilde{\mu}_i(\tilde{\mathbf{z}}_{1:i-1})$
Sampling	slow (sequential)	fast (single pass)
Density estimation	fast (single pass)	slow (sequential)

Lilian Weng, "Flow-based Deep Generative Models," *blog post*, 2018.

Summary

Summary

- Compare different generative models
  - GANs, VAEs and flow-based models
- Survey different normalizing flow models
  - NICE, RealNVP, Glow, MAF and IAF
- Conduct experiments on generating MNIST handwritten digits
  - NICE and RealNVP

Thank you!

[Code] https://github.com/salu133445/flows

[Slides] https://salu133445.github.io/flows