Likelihood based Generative Modeling:

Variational Autoencoders and Normalizing Flows

Seminar on Scientific Machine Learning, Talk 5

Sam Rouppe van der Voort and Elias Huber
under the supervision of Tobias Buck

June 30, 2025

Faculty for Physics and Astronomy
University of Heidelberg

Introduction

Introduction

How can one turn an apple into a banana?

Likelihood based Generative modeling

Likelihood based Generative modeling:
Learn a probability distribution from data, and sample from it.

Problem Setup

Let:

- $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ be observed data.
- $p_\theta(\mathbf{x})$ be a probability density.
- θ be a parameter vector to estimate.

Problem Setup

Let:

- $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ be observed data.
- $p_\theta(\mathbf{x})$ be a probability density.
- θ be a parameter vector to estimate.

Objective: Find θ, $p_\theta(\mathbf{x})$ that model the observed data the best.

Problem Setup

Let:

- $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$ be observed data.
- $p_\theta(\mathbf{x})$ be a probability density.
- θ be a parameter vector to estimate.

Objective: Find θ, $p_\theta(\mathbf{x})$ that model the observed data the best.

Why?

- generate new samples distributed like $p_\theta(\mathbf{x})$
- evaluate data using $p_\theta(\mathbf{x})$.

Likelihood-based Models

Approach

Given data $\mathbf{x}_1, \cdots, \mathbf{x}_n$, we find $p_\theta(x)$ by maximizing the likelihood.

Likelihood:

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p_\theta(\mathbf{x}_i)$$

In practice, we minimize the negative log-likelihood:

$$\hat{\theta}_{MLE} = \arg\min_\theta J(\theta, \mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(n)}) = \frac{1}{n}\sum_{i=1}^{n} -\log p_\theta(\mathbf{x}^{(i)})$$

How?

Key Principle

- We model $p_\theta(\mathbf{x})$ using Neural Networks
- Find optimal θ with Gradient Descent

Generative Modeling

What is Generative Modeling?

Use learned probability density $p_\theta(\mathbf{x})$ to generate new samples
$\rightarrow$ These have same characteristics

Applications:

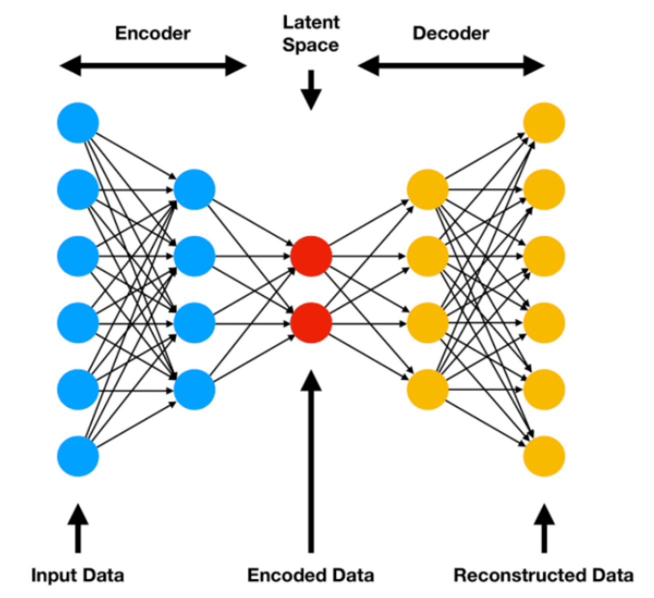
- Image/Video Generation
- Text-to-Anything
- Drug Discovery

Key Approaches:

- Variational Autoencoders (VAEs)
- Normalizing flows

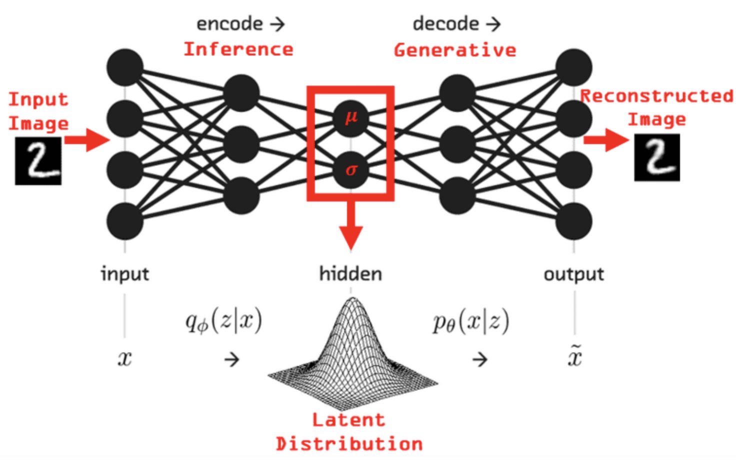
Variational Autoencoders

Variational Autoencoders



https://www.geeksforgeeks.org/numpy-types-of-autoencoders/

Variational Autoencoders



https://theaisummer.com/Autoencoder/

Variational Autoencoders

- Generate $\hat{x}$ following a distribution $p(x)$
- We sample a simple Latent distribution p(z).
  1. $p(x|z)$ as decoder for generation
  2. $p(z|x)$ as encoder for learning latent distribution

VAE Main Idea:

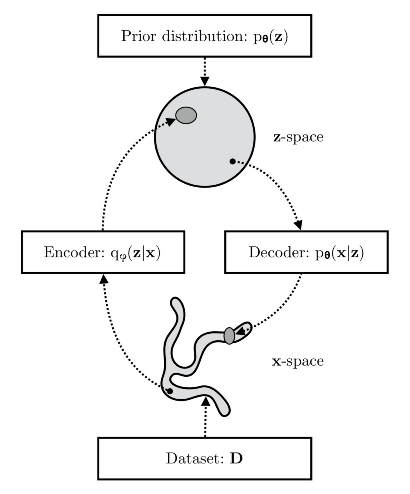
Use a easy to sample from latent distribution $p(z)$ (often Gaussian). To generate new samples $\sim p(x)$. Using approximations of the conditional distributions.

Variational Autoencoders

$\rightarrow$ Why "Variational" AE?

- Not a fixed encoding z, but a distribution over possible z's.
    1. $p(z|x)$ is intractable, so we approximate: $p(z|x) \approx q_\phi(z|x)$.
    2. $p(x|z) \approx p_\theta(x|z)$.
        - Approximate $p_\theta(x|z)$ and $q_\phi(z|x)$ using neural networks.

Variational Autoencoders



[Kingma & Welling, 2019]

vanilla VAE

Main Assumptions:

- latent distribution prior $p(z) \sim \mathcal{N}(0, I)$
- $q_\phi(z|x) \sim \mathcal{N}(\mu_\phi, \sigma_\phi)$. $\rightarrow$ outputs μ_ϕ, σ_ϕ
- $p_\theta(x|z) \sim \mathcal{N}(\mu, \sigma), \text{Bernoulli}(\hat{x})$

VAE objective ELBO

$\rightarrow$ What should our loss function be such that the NNs can approximate the conditional probabilities?

ELBO: Log Likelihood maximization/(minimization)

Maximise $\log p(x) \geq$ Evidence Lower Bound

ELBO

$$
\begin{aligned}
\text{ELBO}(x) &:= \mathbb{E}_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right] \\
&= \mathbb{E}_{q(z|x)}\left[\log p(x|z)\right] - D_{\text{KL}}(q(z|x) \,\|\, p(z))
\end{aligned}
$$

Understanding the ELBO

$$\mathrm{ELBO}(x) = \mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right]$$

$$= \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]}_{\text{Reconstruction}} - \underbrace{D_{\mathrm{KL}}(q_\phi(z|x) \,\|\, p(z))}_{\text{Regularizing}}$$

Understand the ELBO

1. Approximation of the $\log p(x) \rightarrow$ Lower bound

2. $\text{ELBO(x)} = \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]}_{\text{Reconstruction}} - \underbrace{D_{\mathrm{KL}}(q_\phi(z|x) \parallel p(z))}_{\text{Regularizing}}$

ELBO in practice

$$\mathrm{ELBO}(x) = \underbrace{\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x|z)\right]}_{\text{Reconstruction}} - \underbrace{D_{\mathrm{KL}}(q_\phi(z|x) \,\|\, p(z))}_{\text{Regularizing}}$$

$$\underbrace{D_{\mathrm{KL}}(q_\phi(z|x) \,\|\, p(z))}_{\text{Has analytical form.}} \approx \frac{1}{L}\sum_{\ell=1}^{L} \log \frac{q_\phi(z^{(\ell)} \mid x)}{p(z^{(\ell)})}, \quad z^{(\ell)} \sim q_\phi(z \mid x)$$

$$\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] \approx \frac{1}{L}\sum_{\ell=1}^{L} \log p_\theta(x \mid z^{(\ell)}), \quad z^{(\ell)} \sim q_\phi(z \mid x)$$

Gradients of the ELBO

$$\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] \approx \frac{1}{L}\sum_{\ell=1}^{L} \log p_\theta(x \mid z^{(\ell)}), \quad z^{(\ell)} \sim q_\phi(z \mid x)$$

$z^{(\ell)} = q_\phi(z|x) \sim \mathcal{N}(\mu_\phi, \sigma_\phi)$. The sampling is not differentiable w.r.t ϕ.

Gradients of the ELBO

$$\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] \approx \frac{1}{L}\sum_{\ell=1}^{L} \log p_\theta(x \mid z^{(\ell)}), \quad z^{(\ell)} \sim q_\phi(z \mid x)$$

$z^{(\ell)} = q_\phi(z|x) \sim \mathcal{N}(\mu_\phi, \sigma_\phi)$. The sampling is not differentiable w.r.t ϕ.

$\rightarrow$ We reparameterization in terms of a noise parameter $\epsilon \sim \mathcal{N}(0, I)$: $z = g_\phi(\epsilon)$. s.t. it is differential w.r.t ϕ

$\rightarrow z = \mu_\phi + \sigma_\phi \odot \epsilon,$

Gradients of the ELBO: Reparameterization trick

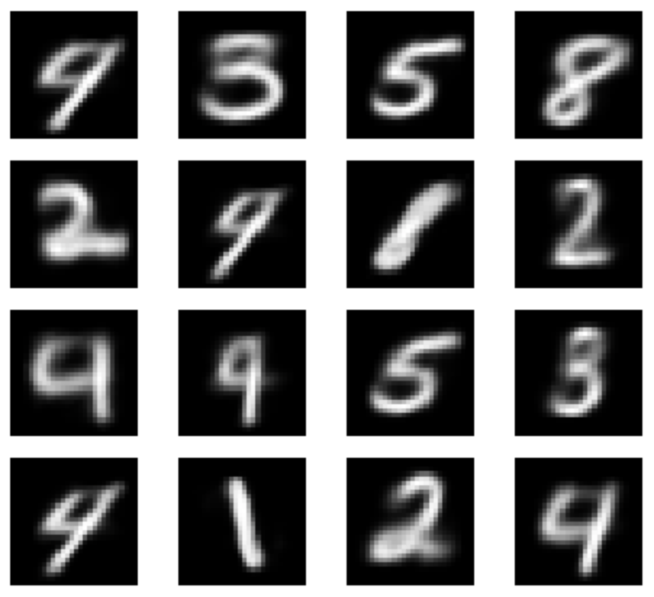
$$\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] \approx \frac{1}{L}\sum_{\ell=1}^{L} \log p_\theta(x \mid z^{(\ell)}), \quad z^{(\ell)} \sim q_\phi(z \mid x)$$

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

$\rightarrow$ To clarify: We move the sampling to $p(\epsilon) \sim \mathcal{N}(0, I)$ such that

$q_\phi(z = \mu_\phi + \sigma_\phi \odot \epsilon|x)$ is differentiable w.r.t μ_ϕ, σ_ϕ.

VAE example MNIST



VAE: Initialization (Encoder, Decoder)

```python
class VAE(nn.Module):
    def __init__(self, input_dim, encoder_hidden_dims, latent_dim, decoder_hidden_dims):
    ↪  (...)
    # === Encoder Part ===
    encoder_layers = []
    in_dim = input_dim
    for h_dim in encoder_hidden_dims:
        encoder_layers.append(nn.Linear(in_dim, h_dim))
        encoder_layers.append(nn.ReLU())
        in_dim = h_dim
    self.encoder_net = nn.Sequential(*encoder_layers)

    self.fc_mu = nn.Linear(in_dim, latent_dim)
    self.fc_log_var = nn.Linear(in_dim, latent_dim)

    # === Decoder Part ===
    decoder_layers = []
    in_dim_decoder = latent_dim
    for h_dim in decoder_hidden_dims:
        decoder_layers.append(nn.Linear(in_dim_decoder, h_dim))
        decoder_layers.append(nn.ReLU())
        in_dim_decoder = h_dim
    self.decoder_net = nn.Sequential(*decoder_layers)

    self.fc_output = nn.Linear(in_dim_decoder, input_dim)
```

VAE: Initialization (Encoder, Decoder)

```
class VAE(nn.Module):
    def __init__(self, input_dim, encoder_hidden_dims, latent_dim, decoder_hidden_dims):
    ↪  (...)
    # === Encoder Part ===
    encoder_layers = []
    in_dim = input_dim
    for h_dim in encoder_hidden_dims:
        encoder_layers.append(nn.Linear(in_dim, h_dim))
        encoder_layers.append(nn.ReLU())
        in_dim = h_dim
    self.encoder_net = nn.Sequential(*encoder_layers)

    self.fc_mu = nn.Linear(in_dim, latent_dim)
    self.fc_log_var = nn.Linear(in_dim, latent_dim)

    # === Decoder Part ===
    decoder_layers = []
    in_dim_decoder = latent_dim
    for h_dim in decoder_hidden_dims:
        decoder_layers.append(nn.Linear(in_dim_decoder, h_dim))
        decoder_layers.append(nn.ReLU())
        in_dim_decoder = h_dim
    self.decoder_net = nn.Sequential(*decoder_layers)

    self.fc_output = nn.Linear(in_dim_decoder, input_dim)
```

VAE: Initialization (Encoder, Decoder)

```
class VAE(nn.Module):
    def __init__(self, input_dim, encoder_hidden_dims, latent_dim, decoder_hidden_dims):
    ↪ (...)
    # === Encoder Part ===
    encoder_layers = []
    in_dim = input_dim
    for h_dim in encoder_hidden_dims:
        encoder_layers.append(nn.Linear(in_dim, h_dim))
        encoder_layers.append(nn.ReLU())
        in_dim = h_dim
    self.encoder_net = nn.Sequential(*encoder_layers)

    self.fc_mu = nn.Linear(in_dim, latent_dim)
    self.fc_log_var = nn.Linear(in_dim, latent_dim)

    # === Decoder Part ===
    decoder_layers = []
    in_dim_decoder = latent_dim
    for h_dim in decoder_hidden_dims:
        decoder_layers.append(nn.Linear(in_dim_decoder, h_dim))
        decoder_layers.append(nn.ReLU())
        in_dim_decoder = h_dim
    self.decoder_net = nn.Sequential(*decoder_layers)

    self.fc_output = nn.Linear(in_dim_decoder, input_dim)
```

VAE: encode and decode Methods

```python
def encode(self, x):
    h_encoder = self.encoder_net(x)
    mu = self.fc_mu(h_encoder)
    log_var = self.fc_log_var(h_encoder)
    return mu, log_var

def decode(self, z):
    h_decoder = self.decoder_net(z)
    recon_x = torch.sigmoid(self.fc_output(h_decoder))
    return recon_x
```

VAE: reparameterize and forward Methods

```python
def reparameterize(self, mu, log_var):
    std = torch.exp(0.5 * log_var)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x):
    x_flat = x.view(-1, self.input_dim)
    mu, log_var = self.encode(x_flat)
    z = self.reparameterize(mu, log_var)
    recon_x = self.decode(z)
    return recon_x, mu, log_var
```

VAE: reparameterize and forward Methods

```python
def reparameterize(self, mu, log_var):
    std = torch.exp(0.5 * log_var)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x):
    x_flat = x.view(-1, self.input_dim)
    mu, log_var = self.encode(x_flat)
    z = self.reparameterize(mu, log_var)
    recon_x = self.decode(z)
    return recon_x, mu, log_var
```

VAE: reparameterize and forward Methods

```python
def reparameterize(self, mu, log_var):
    std = torch.exp(0.5 * log_var)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x):
    x_flat = x.view(-1, self.input_dim)
    mu, log_var = self.encode(x_flat)
    z = self.reparameterize(mu, log_var)
    recon_x = self.decode(z)
    return recon_x, mu, log_var
```

VAE: reparameterize and forward Methods

```python
def reparameterize(self, mu, log_var):
    std = torch.exp(0.5 * log_var)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x):
    x_flat = x.view(-1, self.input_dim)
    mu, log_var = self.encode(x_flat)
    z = self.reparameterize(mu, log_var)
    recon_x = self.decode(z)
    return recon_x, mu, log_var
```

VAE: reparameterize and forward Methods

```python
def reparameterize(self, mu, log_var):
    std = torch.exp(0.5 * log_var)
    eps = torch.randn_like(std)
    return mu + eps * std

def forward(self, x):
    x_flat = x.view(-1, self.input_dim)
    mu, log_var = self.encode(x_flat)
    z = self.reparameterize(mu, log_var)
    recon_x = self.decode(z)
    return recon_x, mu, log_var
```

VAE: Loss

```python
def loss_function(recon_x, x, mu, log_var):
    # 1. Reconstruction Loss
    BCE = F.binary_cross_entropy(recon_x,
    x.view(-1, INPUT_DIM), reduction='sum')

    # 2. KL Divergence
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

    # Total loss
    return BCE + KLD
```

VAE: Loss

```python
def loss_function(recon_x, x, mu, log_var):
    # 1. Reconstruction Loss
    BCE = F.binary_cross_entropy(recon_x,
    x.view(-1, INPUT_DIM), reduction='sum')

    # 2. KL Divergence
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

    # Total loss
    return BCE + KLD
```

VAE: Loss

```python
def loss_function(recon_x, x, mu, log_var):
    # 1. Reconstruction Loss
    BCE = F.binary_cross_entropy(recon_x,
    x.view(-1, INPUT_DIM), reduction='sum')

    # 2. KL Divergence
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

    # Total loss
    return BCE + KLD
```

VAE: Loss

```python
def loss_function(recon_x, x, mu, log_var):
    # 1. Reconstruction Loss
    BCE = F.binary_cross_entropy(recon_x,
    x.view(-1, INPUT_DIM), reduction='sum')

    # 2. KL Divergence
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

    # Total loss
    return BCE + KLD
```

Summary

- VAE $\rightarrow$ AE with latent distribution (easy sampling).
- More continuous latent space than AE
- ELBO loss.
- Reparameterization trick for gradients.
- Gaussian assumption for the prior $p(z)$, and the encoder $q_\phi(z|x)$
- Decoder $p_\theta(x|z)$ assumption based on data.
- [Kingma & Welling, 2019]

Summary

- VAE $\rightarrow$ AE with latent distribution (easy sampling).
- More continuous latent space than AE
- ELBO loss.
- Reparameterization trick for gradients.
- Gaussian assumption for the prior $p(z)$, and the encoder $q_\phi(z|x)$
- Decoder $p_\theta(x|z)$ assumption based on data.
- [Kingma & Welling, 2019]

$\rightarrow$ one Limitation

- inflexibility of gaussian assumption on $q_\phi(z|x)$

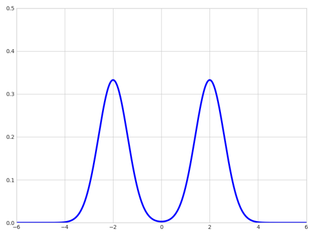
Extensions: more flexible/expressive $q_\phi(z|x)$

1. Normalizing flows
    - More flexible $q(z|x)$ by reparameterization via flows.

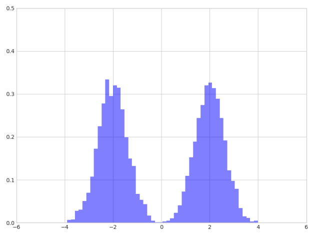
Normalizing flows

Introductory Problem

$p_X(x)$



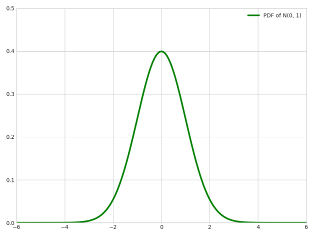
Data following $p_X(x)$



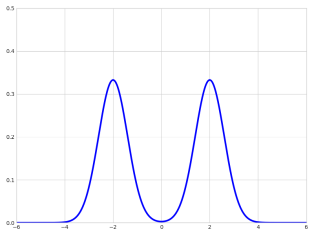
$\rightarrow$ How do we sample or evaluate from p_X?

Introductory Problem

Base distribution: $p_Z(z)$



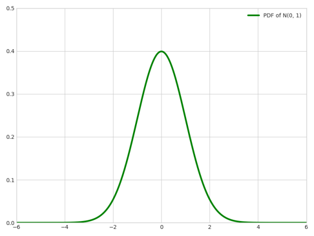
Target distribution: $p_X(x)$



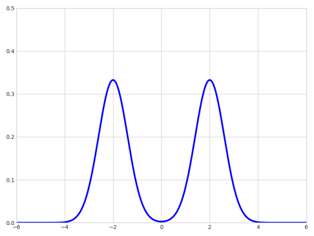
Idea: Change of variables

Introductory Problem

Base distribution: $p_Z(z)$



Target distribution: $p_X(x)$



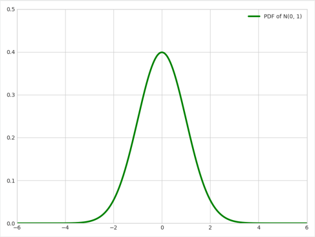
Idea: Change of variables

Solution:

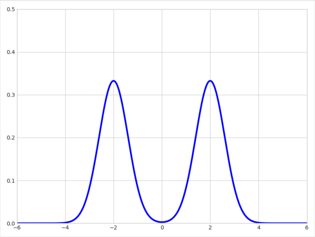
1. Sample from base distribution (e.g., Gaussian) $z \sim p_Z(z)$

Introductory Problem

Base distribution: $p_Z(z)$



Target distribution: $p_X(x)$



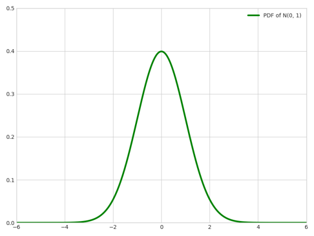
Idea: Change of variables

Solution:

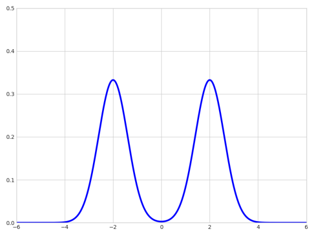
1. Sample from base distribution (e.g., Gaussian) $z \sim p_Z(z)$
2. Apply transformation $x = f^{-1}(z)$

Introductory Problem

Base distribution: $p_Z(z)$



Target distribution: $p_X(x)$



Idea: Change of variables

Solution:

1. Sample from base distribution (e.g., Gaussian) $z \sim p_Z(z)$
2. Apply transformation $x = f^{-1}(z)$
3. Learn f such that x follows desired distribution

Formal Problem Statement

Given:

- A Base distribution $p_Z(\mathbf{z})$ with $\mathbf{z} \in \mathbb{R}^d$
- Data that follows the target $\sim p_X(\mathbf{x})$, with $\mathbf{x} \in \mathbb{R}^d$

Formal Problem Statement

Given:

- A Base distribution $p_Z(\mathbf{z})$ with $\mathbf{z} \in \mathbb{R}^d$
- Data that follows the target $\sim p_X(\mathbf{x})$, with $\mathbf{x} \in \mathbb{R}^d$

Goal:

Find bijective transformation $f : \mathbb{R}^d \to \mathbb{R}^d$, such that

$$p_X(\mathbf{x}) = p_Z(f(\mathbf{x})) \left|\det J_f(\mathbf{x})\right|^{-1}$$

Formal Problem Statement

Given:

- A Base distribution $p_Z(\mathbf{z})$ with $\mathbf{z} \in \mathbb{R}^d$
- Data that follows the target $\sim p_X(\mathbf{x})$, with $\mathbf{x} \in \mathbb{R}^d$

Goal:

Find bijective transformation $f : \mathbb{R}^d \to \mathbb{R}^d$, such that

$$p_X(\mathbf{x}) = p_Z(f(\mathbf{x})) \left|\det J_f(\mathbf{x})\right|^{-1}$$

$\rightarrow$ This is just a coordinate transform

Formal Problem Statement

Given:

- A Base distribution $p_Z(\mathbf{z})$ with $\mathbf{z} \in \mathbb{R}^d$
- Data that follows the target $\sim p_X(\mathbf{x})$, with $\mathbf{x} \in \mathbb{R}^d$

Goal:

Find bijective transformation $f : \mathbb{R}^d \to \mathbb{R}^d$, such that

$$p_X(\mathbf{x}) = p_Z(f(\mathbf{x})) \left|\det J_f(\mathbf{x})\right|^{-1}$$

$\rightarrow$ This is just a coordinate transform

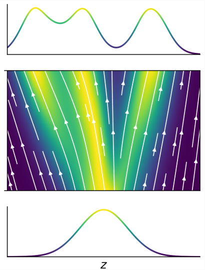
Then:

- evaluate the probability $p_X(\mathbf{x})$
- generate new samples $\mathbf{x} = f^{-1}(\mathbf{z}) \sim p_X(x)$

Designing Neural Networks for Normalizing flows

Idea:

Learn $f(z)$ with Neural Networks.
$\rightarrow$ We don't learn the distribution, **but** the transformation

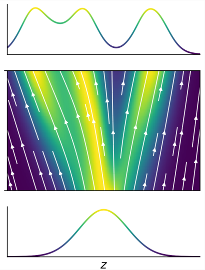


[Grathwohl et al., 2018]

Designing Neural Networks for Normalizing flows

Idea:

Learn $f(z)$ with Neural Networks.
$\rightarrow$ We don't learn the distribution, **but** the transformation



[Grathwohl et al., 2018]

Key considerations:

We need to be able to compute

- $\mathbf{x} = f^{-1}(\mathbf{z})$
- $|\det J_f(\mathbf{x})|^{-1}$

Designing Neural Networks for Normalizing flows

The Neural Network

- must be invertible and differentiable
- has an efficiently computable inverse and Jacobian determinant

Designing Neural Networks for Normalizing flows

The Neural Network

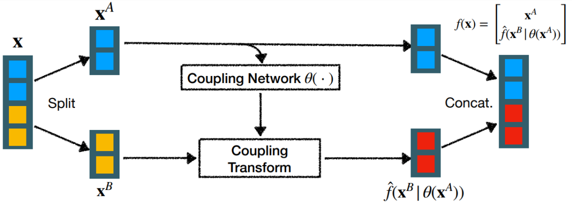
- must be invertible and differentiable
- has an efficiently computable inverse and Jacobian determinant

We trade-off between:

- Expressivity
- Computational speed

Coupling Flows: Partition-based Invertible Transformations

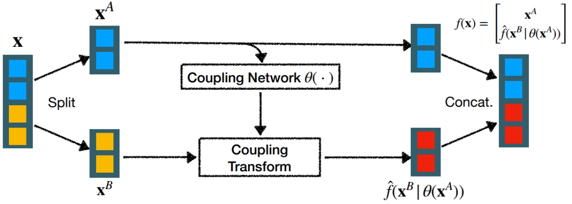
Key Idea: Split input $\mathbf{x} = [\mathbf{x}_A, \mathbf{x}_B]$ and transform only one part



[Brubaker and Köthe, 2021]

Coupling Flows: Partition-based Invertible Transformations

Key Idea: Split input $\mathbf{x} = [\mathbf{x}_A, \mathbf{x}_B]$ and transform only one part

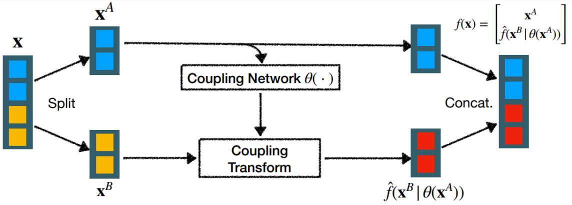


[Brubaker and Köthe, 2021]

$$J_f(\mathbf{x}) = \begin{bmatrix} I & 0 \\ \frac{\partial}{\partial \mathbf{x}_A} \hat{f}(\mathbf{x}_B \mid \theta(\mathbf{x}_A)) & D\hat{f}(\mathbf{x}_B \mid \theta(\mathbf{x}_A)) \end{bmatrix}$$

Coupling Flows: Partition-based Invertible Transformations

Key Idea: Split input $\mathbf{x} = [\mathbf{x}_A, \mathbf{x}_B]$ and transform only one part



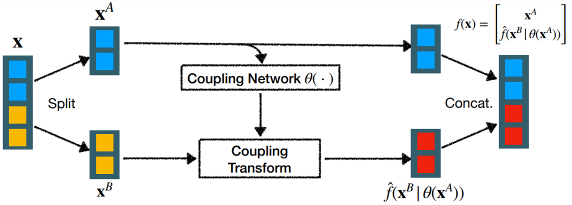
[Brubaker and Köthe, 2021]

$$J_f(\mathbf{x}) = \begin{bmatrix} I & 0 \\ \frac{\partial}{\partial \mathbf{x}_A} \hat{f}(\mathbf{x}_B \mid \theta(\mathbf{x}_A)) & D\hat{f}(\mathbf{x}_B \mid \theta(\mathbf{x}_A)) \end{bmatrix}$$

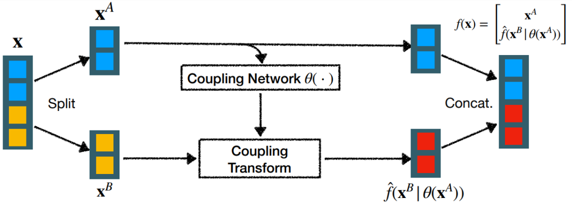
Advantages:

- Efficient Jacobian, Determinant computation

Coupling Flows: Partition-based Invertible Transformations



Coupling Flows: Partition-based Invertible Transformations



$\rightarrow$ What about the first dimensions?

Composing flows

We can compose flows:

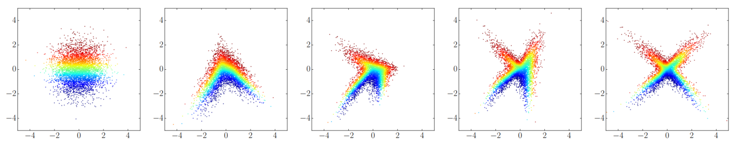
$$f = f_K \circ f_{K-1} \circ \cdots \circ f_1$$

Composing flows

We can compose flows:

$$f = f_K \circ f_{K-1} \circ \cdots \circ f_1$$

$\rightarrow$ This enables the division of a flow into simple subflows



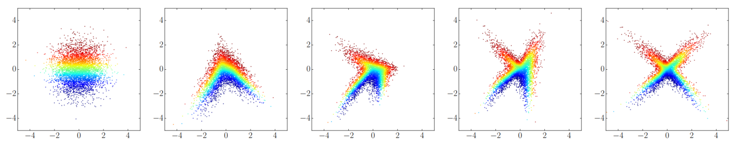
[Papamakarios et al., 2021]

Composing flows

We can compose flows:

$$f = f_K \circ f_{K-1} \circ \cdots \circ f_1$$

$\rightarrow$ This enables the division of a flow into simple subflows



[Papamakarios et al., 2021]

The determinant then yields:

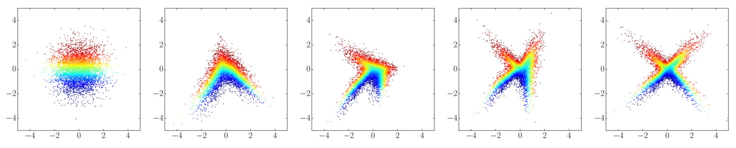
$$\det J_f = \det J_{f_K} \cdot \det J_{f_{K-1}} \cdots \det J_{f_1} = \prod_{k=1}^{K} \det J_{f_k}$$

Composing flows

We can compose flows:

$$f = f_K \circ f_{K-1} \circ \cdots \circ f_1$$

$\rightarrow$ This enables the division of a flow into simple subflows



[Papamakarios et al., 2021]

The determinant then yields:

$$\det J_f = \det J_{f_K} \cdot \det J_{f_{K-1}} \cdots \det J_{f_1} = \prod_{k=1}^{K} \det J_{f_k}$$

$\rightarrow$ Transform different dimensions during each subflow

Training Normalizing Flows

- Typically trained via maximum likelihood:

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim p_{data}}[\log p_\theta(\mathbf{x})]$$

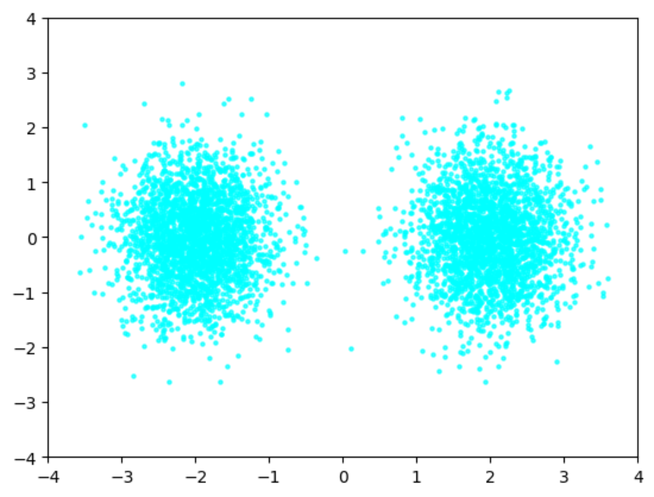
- For flows, this becomes:

$$\mathcal{L}(\theta) = -\mathbb{E}_{\mathbf{x} \sim p_{data}}[\log(\underbrace{p_Z(f^{-1}(\mathbf{x};\theta)}_{\text{Transformation error}})) + \log \underbrace{|\det J_{f^{-1}}(\mathbf{x};\theta)|}_{\text{Normalization}}]$$

Summary Normalizing Flows

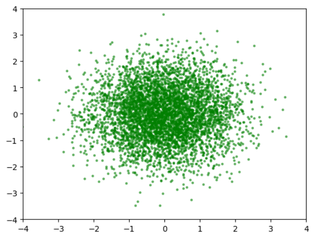
- Use Neural Networks to learn transforms between distributions
- From simple to complicated distribution
- NN has to be invertible, differentiable

Example: Learn a double Gaussian distribution flow

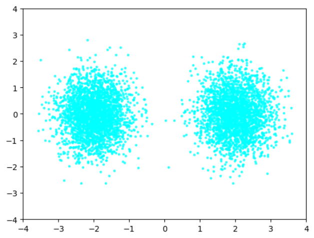


Example: Learn a double Gaussian distribution flow

Base distribution: $p_Z(z)$



Target distribution: $p_X(x)$



Example: Learn a double Gaussian distribution flow

```python
def create_normalizing_flow(num_layers=8, hidden_features=128):
    base_dist = StandardNormal(shape=[2])
    transforms = []
    for i in range(num_layers):
        mask = torch.zeros(2); mask[i % 2] = 1
        transforms.append(
            AffineCouplingTransform(
                mask=mask,
                transform_net_create_fn=lambda in_feat, out_feat: ContextNet(in_feat, out_feat, hidden_features)
            )
        )
    transform = CompositeTransform(transforms)
    flow = Flow(transform, base_dist)
    return flow
```

Example: Learn a double Gaussian distribution flow

```
def create_normalizing_flow(num_layers=8, hidden_features=128):
    base_dist = StandardNormal(shape=[2])
    transforms = []
    for i in range(num_layers):
        mask = torch.zeros(2); mask[i % 2] = 1
        transforms.append(
            AffineCouplingTransform(
                mask=mask,
                transform_net_create_fn=lambda in_feat, out_feat:
                ↪  ContextNet(in_feat, out_feat, hidden_features)
            )
        )
    transform = CompositeTransform(transforms)
    flow = Flow(transform, base_dist)
    return flow
```

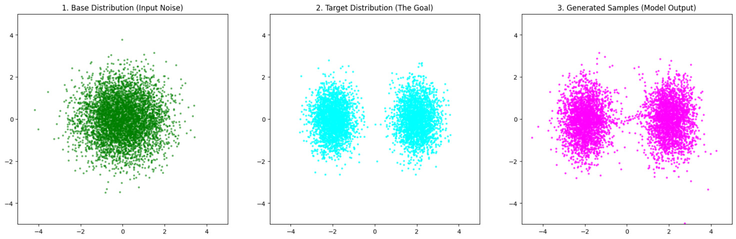
Example: Learn a double Gaussian distribution flow

```python
model.train()
pbar = tqdm(range(num_iter), desc="Training NF")
for i in pbar:
    x = mixture_dist.sample((batch_size,))
    loss = -model.log_prob(inputs=x).mean()
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Example: Learn a double Gaussian distribution flow

```
model.train()
pbar = tqdm(range(num_iter), desc="Training NF")
for i in pbar:
    x = mixture_dist.sample((batch_size,))
    loss = -model.log_prob(inputs=x).mean()
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Example: Learn a double Gaussian distribution flow



Challenges and Limitations

- Computational cost of Jacobian determinant calculations
- Trade-off between expressiveness and fast Invertability
- Often outperformed by other methods

$\rightarrow$ Can we improve Normalizing Flows?

Continuous Normalizing Flows: From Discrete to Continuous Transformations

Discrete NFs

- K fixed transformations
- $\mathbf{z}_k = f_k(\mathbf{z}_{k-1})$

Continuous NFs

- Continuous trajectory $\mathbf{z}(t)$
- $\frac{d\mathbf{z}}{dt} = f(\mathbf{z}(t), t; \theta)$

Key Insight: Replace discrete steps with continuous-time ODE

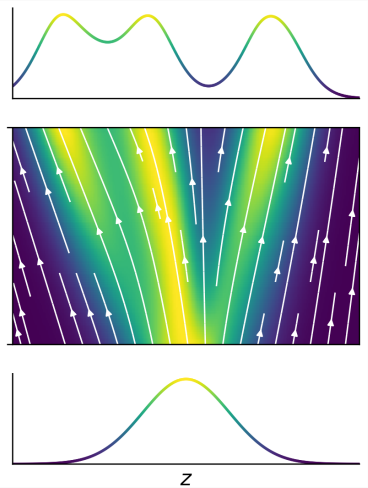
Connection to Neural ODEs

- Dynamics are described by an ODE:

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t; \theta)$$

  where f is a neural network

- Key advantages:
  - no need for matrix inversion
  - more flexible



[Grathwohl et al., 2018]

Example 2: Learn distribution using flow-matching

```
for epoch in tqdm(range(NUM_EPOCHS)):
    model.train()

    # Sample data from base distribution
    x = torch.randn(BATCH_SIZE, 2) * STD
    # Sample from target distribution
    y = qr_dist.sample(BATCH_SIZE)
    t = torch.rand(BATCH_SIZE, 1)

    # Perform flow matching
    psi_t = (1 - t) * x + t * y
    v_pred = model(psi_t, t.squeeze())
    v_true = y - x

    # Loss and backprop
    loss = criterion(v_pred, v_true)
    optimizer.zero_grad()
    loss.backward()
```

Example 2: Learn distribution using flow-matching

```
for epoch in tqdm(range(NUM_EPOCHS)):
    model.train()

    # Sample data from base distribution
    x = torch.randn(BATCH_SIZE, 2) * STD
    # Sample from target distribution
    y = qr_dist.sample(BATCH_SIZE)
    t = torch.rand(BATCH_SIZE, 1)

    # Perform flow matching
    psi_t = (1 - t) * x + t * y
    v_pred = model(psi_t, t.squeeze())
    v_true = y - x

    # Loss and backprop
    loss = criterion(v_pred, v_true)
    optimizer.zero_grad()
    loss.backward()
```

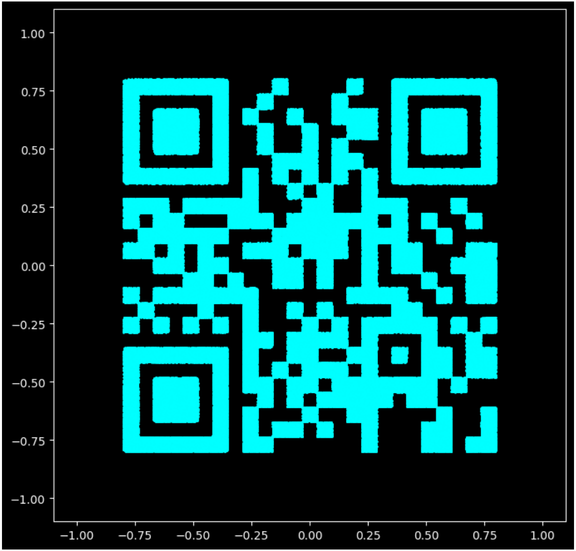
Example 2: Learn distribution using flow-matching

```
for epoch in tqdm(range(NUM_EPOCHS)):
    model.train()

    # Sample data from base distribution
    x = torch.randn(BATCH_SIZE, 2) * STD
    # Sample from target distribution
    y = qr_dist.sample(BATCH_SIZE)
    t = torch.rand(BATCH_SIZE, 1)

    # Perform flow matching
    psi_t = (1 - t) * x + t * y
    v_pred = model(psi_t, t.squeeze())
    v_true = y - x

    # Loss and backprop
    loss = criterion(v_pred, v_true)
    optimizer.zero_grad()
    loss.backward()
```

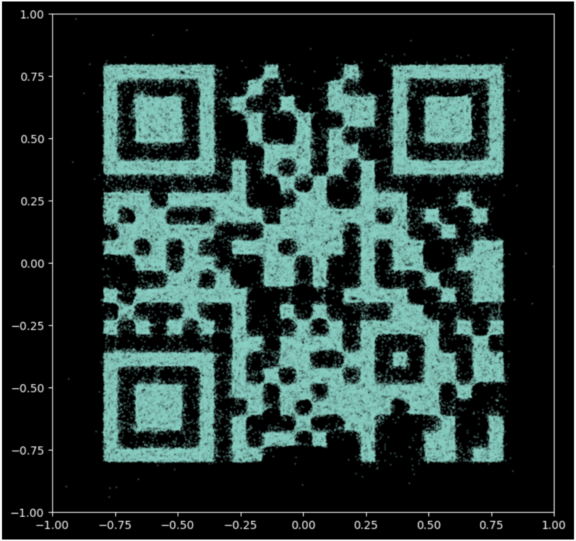
Example: Learn a QR-Code distribution flow

```
# Initialize from Gaussian prior
x = torch.randn(current_batch_size, 2, device=device) * BASE_DISTRIBUTION_STD

# Euler integration
for step in range(num_steps):
    t = torch.full((current_batch_size,), step/num_steps)
    dx = model(x, t) / num_steps
    x = x + dx
```

Example 2: Learn distribution using flow-matching



Example: Learn a QR-Code distribution flow



Applications in Physics

General Application Fields

Both architectures can be used in two ways:

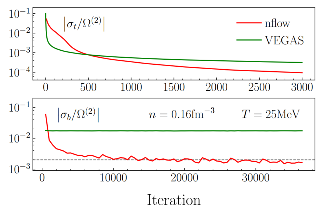
- backward (computing the likelihood of a sample)
- forward (sampling from the modeled distribution)

- **Generative modeling**: High-quality image generation
- **Density estimation**: Anomaly detection, Monte Carlo sampling

Normalizing Flows in Physics: Example

Application in Monte Carlo Sampling:

- Perform Monte Carlo importance sampling for high-dimensional integrals
- The flow learns the probability distribution necessary for MC integration, enabling more precise and efficient sampling



[Roggero et al., 2021]

VAEs in Physics

- Galaxy generation for calibration [Ravanbakhsh et al., 2016]
- Chemical design [Gomez-Bombarelli et al., 2018]
- Anomaly detection at LHC [Pol et al., 2020]
- Multimodal data integration Time Series [Brenner et al., 2024]

The Banana Problem

But how do we turn an apple into a banana?

The Banana Problem

1. Get Fruit dataset

The Banana Problem

1. Get Fruit dataset
2. Use VAE to learn Fruit distributions
    - Optimize ELBO
    - with reparameterization trick

The Banana Problem

1. Get Fruit dataset
2. Use VAE to learn Fruit distributions
    - Optimize ELBO
    - with reparameterization trick
3. Improve the flexibility of the encoder by applying normalizing flows on the latent space.

The Banana Problem

1. Get Fruit dataset
2. Use VAE to learn Fruit distributions
    - Optimize ELBO
    - with reparameterization trick
3. Improve the flexibility of the encoder by applying normalizing flows on the latent space.
    - Create complicated distribution from simple Gaussian

The Banana Problem

1. Get Fruit dataset
2. Use VAE to learn Fruit distributions
    - Optimize ELBO
    - with reparameterization trick
3. Improve the flexibility of the encoder by applying normalizing flows on the latent space.
    - Create complicated distribution from simple Gaussian
    - NN has to be invertible and differentiable

The Banana Problem

1. Get Fruit dataset
2. Use VAE to learn Fruit distributions
   - Optimize ELBO
   - with reparameterization trick
3. Improve the flexibility of the encoder by applying normalizing flows on the latent space.
   - Create complicated distribution from simple Gaussian
   - NN has to be invertible and differentiable
4. Construct Apple to Banana path in latent space

The Banana Problem

1. Get Fruit dataset
2. Use VAE to learn Fruit distributions
    - Optimize ELBO
    - with reparameterization trick
3. Improve the flexibility of the encoder by applying normalizing flows on the latent space.
    - Create complicated distribution from simple Gaussian
    - NN has to be invertible and differentiable
4. Construct Apple to Banana path in latent space
5. Decode alongside the latent space path

The Banana Problem

1. Get Fruit dataset
2. Use VAE to learn Fruit distributions
    - Optimize ELBO
    - with reparameterization trick
3. Improve the flexibility of the encoder by applying normalizing flows on the latent space.
    - Create complicated distribution from simple Gaussian
    - NN has to be invertible and differentiable
4. Construct Apple to Banana path in latent space
5. Decode alongside the latent space path
6. enjoy!

References

Durkan, Conor and Bekasov, Artur and Murray, Iain and Papamakarios, George (2019)
Neural Spline Flows
Advances in Neural Information Processing Systems 32, 7511–7522.

Papamakarios, George and Nalisnick, Eric and Jimenez Rezende, Danilo and Mohamed, Shakir and Lakshminarayanan, Balaji (2021)
Normalizing Flows for Probabilistic Modeling and Inference
Journal of Machine Learning Research 22(57), 1–64.

Kingma, Diederik P. and Salimans, Tim and Jozefowicz, Rafal and Chen, Xi and Sutskever, Ilya and Welling, Max (2016)
Improving Variational Inference with Inverse Autoregressive Flow
arXiv preprint arXiv:1606.04934.

Kobyzev, Ivan and Prince, Simon J.D. and Brubaker, Marcus A. (2021)
Normalizing Flows: An Introduction and Review of Current Methods
IEEE Transactions on Pattern Analysis and Machine Intelligence 43(11), 3964–3979.

References iii

PapersWithCode (2025)
An Overview of Likelihood-Based Generative Models
Online resource.

Kingma, Diederik P. and Welling, Max (2019)
An Introduction to Variational Autoencoders
arXiv preprint arXiv:1906.02691.

Chen, Yaoyu (2019)
Likelihood-based Generative Models I: Autoregressive Models
Personal blog post.

Brenner, Manuel and Hess, Florian and Koppe, Georgia and Durstewitz, Daniel (2024)
Integrating Multimodal Data for Joint Generative Modeling of Complex Dynamics
Proceedings of the 41st International Conference on Machine Learning 235, 4482–4516.

Grathwohl, Will and Chen, Ricky T. Q. and Bettencourt, Jesse and Sutskever, Ilya and Duvenaud, David (2018)
FFJORD: Free-form Continuous Dynamics for Scalable Reversible Generative Models.
arXiv preprint arXiv:1810.01367.
https://arxiv.org/abs/1810.01367

Roggero, Alessandro and Hackstein, Christian and Carlson, Joseph (2021)

Normalizing Flows for Microscopic Many-Body Calculations: An Application to the Nuclear Equation of State.

Physical Review Letters 127(6), 062502.

https://www.osti.gov/pages/biblio/1850367

Gomez-Bombarelli, Rafael and Wei, Jennifer N. and Du, David and Gimeno, Miquel and Zepeda-Cervantes, Jorge and ... and Aspuru-Guzik, Alán (2018)

Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules

ACS Central Science 4(2), 268–276.
https://arxiv.org/abs/1610.02415

Ravanbakhsh, Siamak and Lanusse, Francois and Mandelbaum, Rachel and Schneider, Jeff and Poczos, Barnabas (2016)
Enabling Dark Energy Science with Deep Generative Models of Galaxy Images
arXiv preprint arXiv:1609.05796.
https://arxiv.org/abs/1609.05796

Pol, Adrian Alan and Berger, Victor and Cerminara, Gianluca and Germain, Cécile and Pierini, Maurizio (2020)
Anomaly Detection With Conditional Variational Autoencoders
arXiv preprint arXiv:2010.05531.
https://arxiv.org/abs/2010.05531

Brubaker, Marcus A. and Köthe, Ullrich (2021)
Normalizing Flows and Invertible Neural Networks in Computer Vision: CVPR 2021 Tutorial.
CVPR 2021 Tutorial.
https://mbrubake.github.io

Backup slides

Other flows

Most Widely Used Flow Types

Coupling Flows:

- split data to ensure a triangular Jacobian matrix

Autoregressive Flows:

- Each input dimension is transformed, only depending on the previous ones, making the Jacobian triangular by design

Spline-based Flows:

- Use piecewise polynomial functions
- More expressive than affine transformations

Extra slide: Understanding the ELBO

$$\mathrm{ELBO}(x,z) = \mathbb{E}_{q(z|x)}\left[\log \frac{p(x,z)}{q(z|x)}\right] = \mathbb{E}_{q(z|x)}\left[\log \frac{p(z|x)p(x)}{q(z|x)}\right]$$

$$= \underbrace{\mathbb{E}_{q(z|x)}\left[\log p(x)\right]}_{\text{Estimate of log likelihood.}} - D_{\mathrm{KL}}(q(z|x) \parallel p(z|x))$$

$$\mathrm{ELBO}(x,z) \leq \log p(x)$$

Extra slide: More details on Reparameterization trick

$$\mathbb{E}_{q_\phi(z|x)}\left[\log p_\theta(x \mid z)\right] \approx \frac{1}{L}\sum_{\ell=1}^{L} \log p_\theta(x \mid z^{(\ell)}), \quad z^{(\ell)} \sim q_\phi(z \mid x)$$

$z^{(\ell)} = q_\phi(z|x) \sim \mathcal{N}(\mu_\phi, \sigma_\phi)$. The sampling is not differentiable w.r.t ϕ.

$\rightarrow$ We reparameterize in terms of a noise parameter $\epsilon \sim \mathcal{N}(0, I)$:

$z = g_\phi(\epsilon)$. s.t. it is differential w.r.t ϕ

$$\rightarrow z = \mu_\phi + \sigma_\phi \odot \epsilon,$$

$$q_\phi(z \mid x) = p(g_\phi^{-1}(z)) \cdot \left|\det\left(\frac{\partial g_\phi(\epsilon)}{\partial \epsilon}\right)\right|^{-1}$$

$$\log q_\phi(z \mid x) = \log p(\epsilon) - \log\left|\det\left(\frac{\partial z}{\partial \epsilon}\right)\right|$$

Extra Slide: More in depth example: VAEs DSR

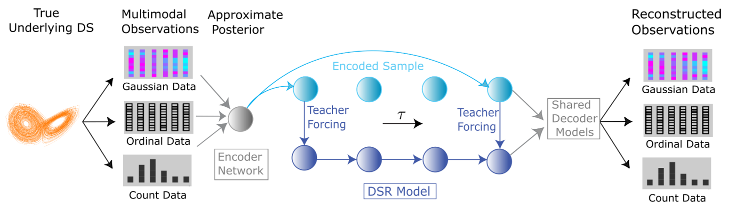


Figure 1. MTF setup. Multimodal observations are translated via an encoder into a common latent representation, which is used for sparse TF in the DSR model's latent space. The latent trajectory is then mapped back into observation space via modality-specific decoder models, which are shared between the MVAE and DSR model.

[Brenner et al., 2024]