Generative Models

in HEP

ML4FP School 2026 — Georgia Tech

Sofia Palacios Schweitzer, Rutgers University

spalacios@physics.rutgers.edu

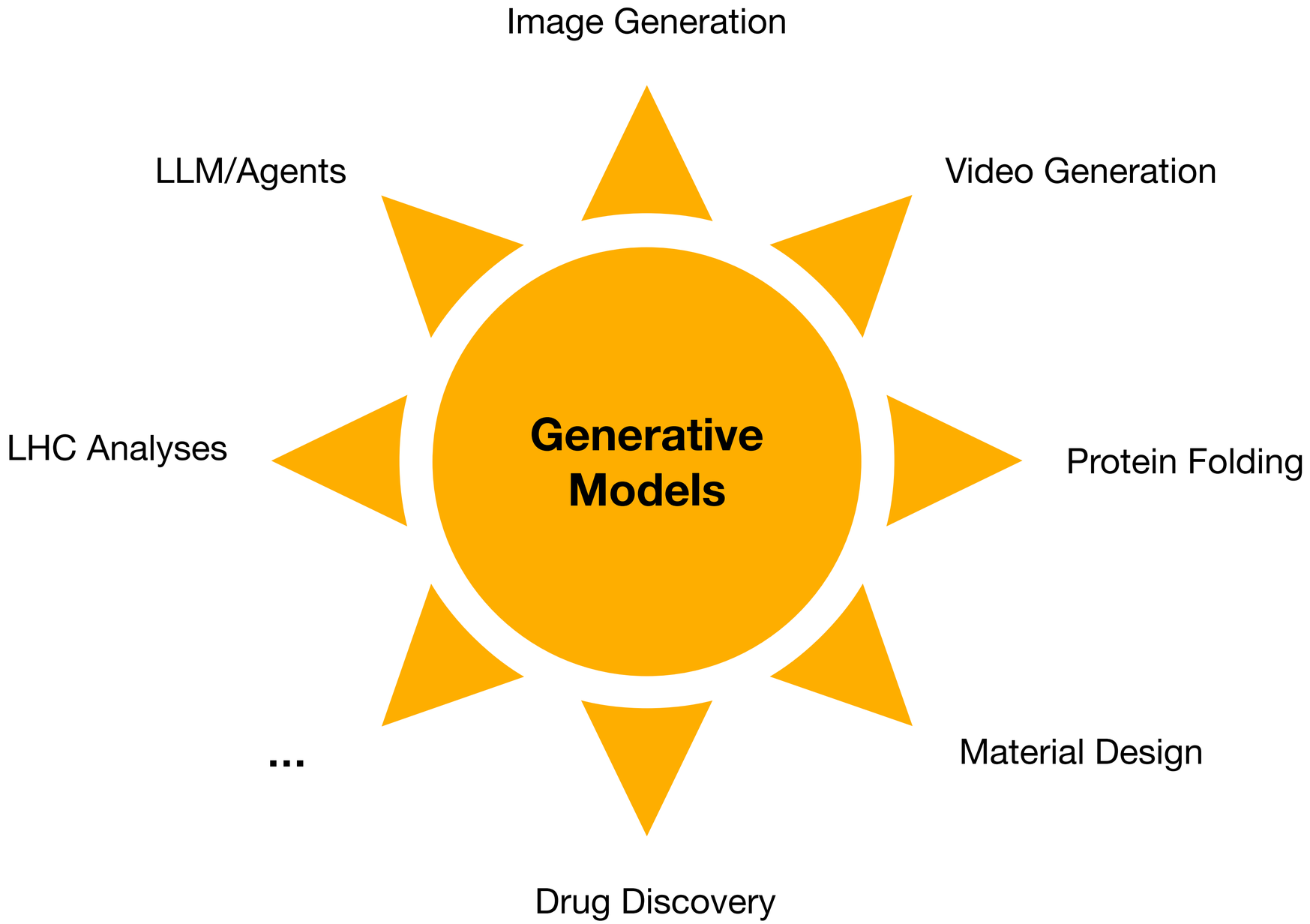


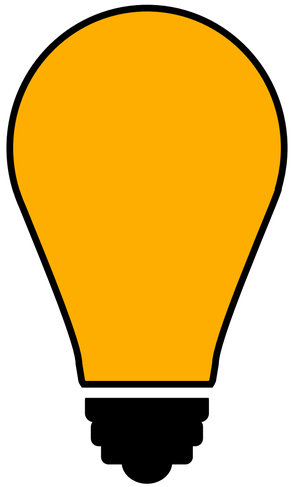
Image Generation
LLM/Agents
Video Generation
Generative Models
LHC Analyses
Protein Folding
...
Material Design
Drug Discovery

How to make us of generative AI?

1. Understanding the underlying algorithms

2. Finding the right problem formulation

3. Validating generated samples & verifying solution

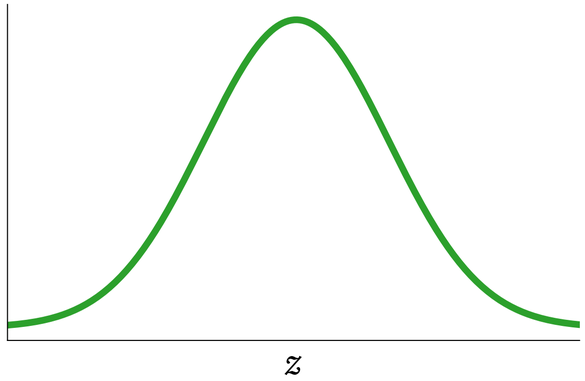


1. Understanding the underlying algorithms

Learning a transformation

Latent space

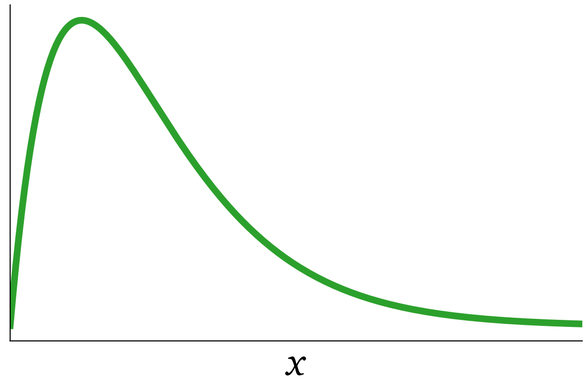
$p(z)$



?

Phase space

$p(x)$



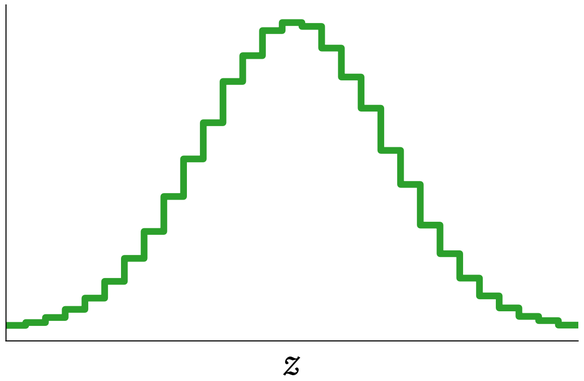
- easy to sample from
- (sometimes) known analytical form
- usually not physically interpretable

- hard to sample from
- (usually) unknown analytical form
- physically interpretable

Learning a transformation

Latent space

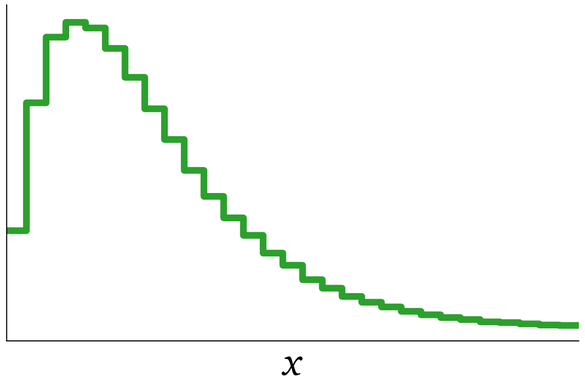
$z \sim p(z)$



- easy to sample from
- (sometimes) known analytical form
- usually not physically interpretable

Phase space

$x \sim p(x)$

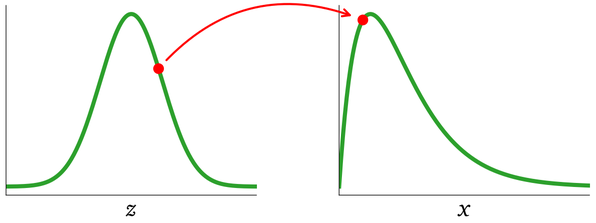


- hard to sample from
- (usually) unknown analytical form
- physically interpretable

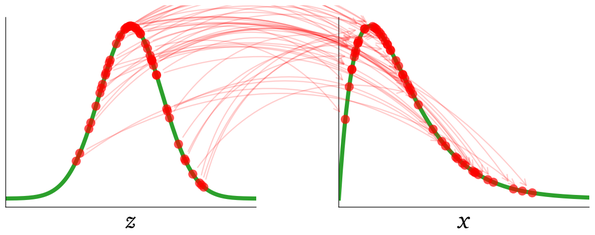
Learning a transformation

Requirements

1. $z \sim p(z) \xrightarrow{\text{transform}} x \sim p(x)$



2. $p_{\text{model}}(x) = p(x)$

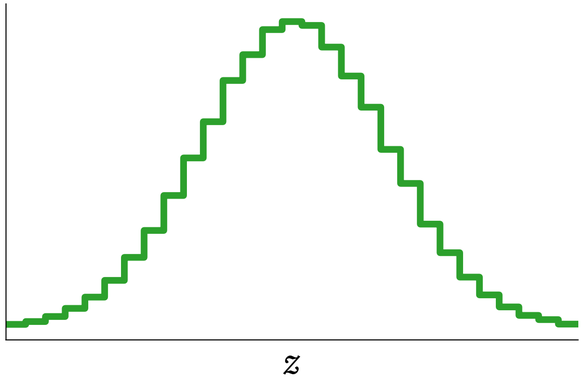


3. Feasible training & inference time

Learning a transformation

Latent space

$z \sim p(z)$

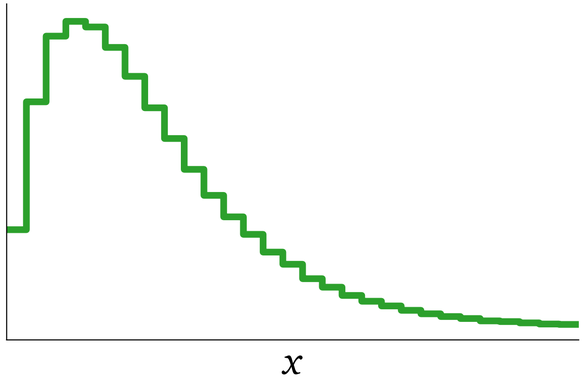


?

A Neural Network?

Phase space

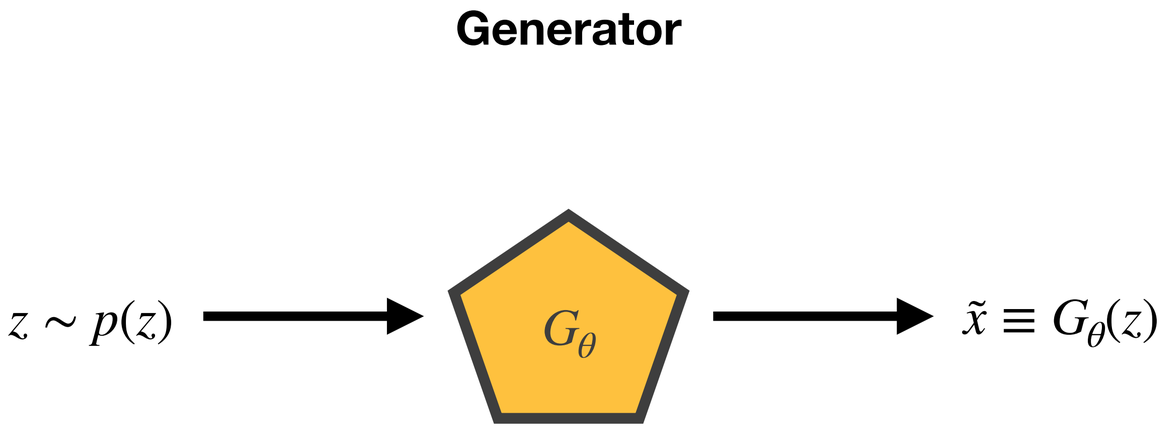
$x \sim p(x)$



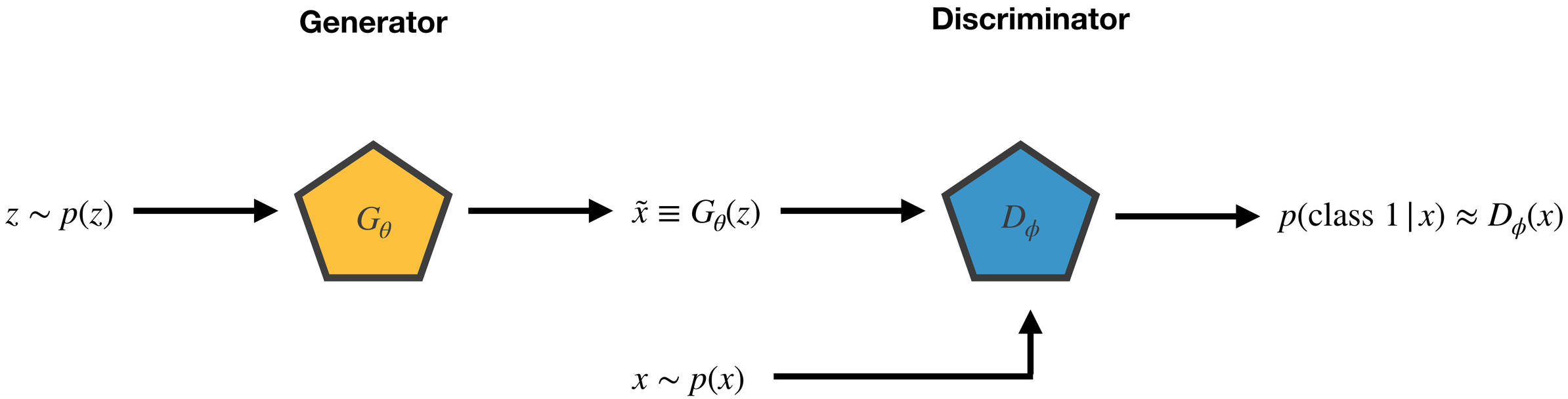
- easy to sample from
- (sometimes) known analytical form
- usually not physically interpretable

- hard to sample from
- (usually) unknown analytical form
- physically interpretable

Generative Adversarial Networks
(GANs)



Generative Adversarial Networks
(GANs)



Generative Adversarial Networks
(GANs)

Discriminator

Classification task: Distinguish class 1 (true) samples from class 0 (generated) samples

$$-\int dz\, p(z)\ \log\Big(1 - D_\phi(G_\theta(z))\Big) - \int dx\, p(x)\ \log D_\phi(x)$$

Minimal for $D_\phi(G_\theta(z)) \to 0$

Minimal for $D_\phi(x) \to 1$

Generator

Goal: Fool discriminator $p(\text{class } 1 \,|\, G_\theta(z)) \to 1$

$$\int dz\, p(z)\ \log\Big(1 - D_\phi(G_\theta(z))\Big)$$

Minimal for $D_\phi(G_\theta(z)) \to 1$

Training

```python
def batch_loss(self, x):
    """
    Args:
        x: real samples from phase space, shape (batch_size, ...)

    Returns:
        d_loss, g_loss: discriminator and generator losses
    """
    batch_size = x.size(0)
    z = self.latent.sample((batch_size,)).to(x.device)

    real_labels = torch.ones(batch_size, 1, device=x.device)
    fake_labels = torch.zeros(batch_size, 1, device=x.device)

    x_fake = self.generator(z)

    # ---- Discriminator loss ----
    # D maximizes  E[log D(x)] + E[log(1 - D(G(z)))]
    d_real = self.discriminator(x)
    d_fake = self.discriminator(x_fake.detach())  # detach: no gradients to G here

    d_loss = F.binary_cross_entropy(d_real, real_labels) \
           + F.binary_cross_entropy(d_fake, fake_labels)

    # ---- Generator loss (saturating, original form) ----
    # G minimizes  E[log(1 - D(G(z)))]
    d_gen = self.discriminator(x_fake)
    g_loss = torch.log(1.0 - d_gen).mean()

    return d_loss, g_loss
```

Training

```
for x in dataloader:
    d_loss, g_loss = self.batch_loss(x)

    self.opt_d.zero_grad()
    d_loss.backward()
    self.opt_d.step()

    self.opt_g.zero_grad()
    g_loss.backward()
    self.opt_g.step()
```

Sampling

```
def sample(self, n):
    """
    Generate n samples from the model.

    Args:
        n: number of samples to draw

    Returns:
        x_gen: generated samples, shape (n, ...)
    """
    self.generator.eval()
    with torch.no_grad():
        z = self.latent.sample((n,))
        x_gen = self.generator(z)
    return x_gen
    self.opt_g.step()
```

Generative Adversarial Networks
(GANs)

$$\mathscr{L}_{\text{GAN}} = \max_{\phi} \min_{\theta} \int dz\, p(z)\ \log\left(1 - D_{\phi}(G_{\theta}(z))\right) + \int dx\, p(x)\ \log D_{\phi}(x)$$

In summary

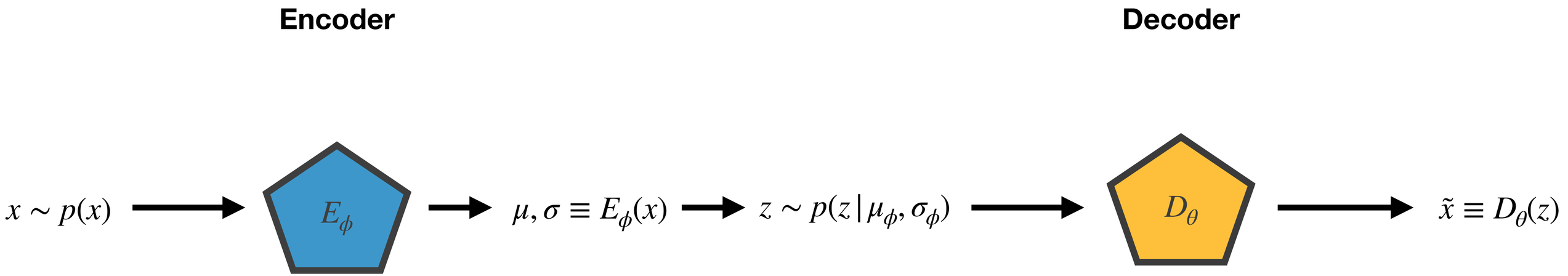
1. Two arbitrary networks G_{θ}, D_{ϕ}
2. An arbitrary latent space of arbitrary dimensionality $p(z)$
3. Single-shot sampling

What's the problem?

1. Unstable
2. Mode collapse
3. No direct access to likelihood

Variational Autoencoders
(VAEs)

Variational Autoencoders
(VAEs)



Variational Autoencoders
(VAEs)

Encoder

Encode x into Gaussian latent space

$$z = \mu_\phi(x) + \sigma_\phi(x)\epsilon$$

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

Effectively

$$z \sim p(z \mid \mu_\phi(x), \sigma_\phi(x)) \equiv p_\phi(z \mid x)$$

Decoder

Decode z into phase space

$$x = D_\theta(z) + \sigma\epsilon$$

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

Effectively

$$x \sim p(x \mid D_\theta(z)) \equiv p_\theta(x \mid z)$$

Variational Autoencoders
(VAEs)

Encoder

Encode x into Gaussian latent space

$$z = \mu_\phi(x) + \sigma_\phi(x)\epsilon \sim p_\phi(z\,|\,x)$$

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

$$\underbrace{D_{KL}(p_\phi(z\,|\,x)\|\ \mathcal{N}(0,1))}$$

Regularisation

Decoder

Decode z into phase space

$$x = D_\theta(z)$$

$$\underbrace{\|x - D_\theta(z)\|^2}$$

Reconstruction

Training

```
def batch_loss(self, x):
    """
    Args:
        x: real samples from phase space, shape (batch_size, ...)

    Returns:
        loss: total VAE loss (reconstruction + KL)
    """
    # Encode: returns parameters of p_phi(z|x) = N(mu, sigma^2)
    mu, log_var = self.encoder(x)

    # Reparameterization: z = mu + sigma * eps,  eps ~ N(0, I)
    eps = torch.randn_like(mu)
    z = mu + torch.exp(0.5 * log_var) * eps

    # Decode
    x_recon = self.decoder(z)

    # ---- Reconstruction ----
    # MSE between x and decoder output
    recon_loss = ((x - x_recon) ** 2).sum(dim=1).mean()

    # ---- Latent regularization ----
    # KL( N(mu, sigma^2) || N(0, I) ), closed form
    kl_loss = 0.5 * (mu.pow(2) + log_var.exp() - log_var - 1).sum(dim=1).mean()

    return recon_loss + kl_loss
```

Sampling

```python
def sample(self, n):
    """
    Generate n samples from the model.

    Args:
        n: number of samples to draw

    Returns:
        x_gen: generated samples, shape (n, ...)
    """
    self.decoder.eval()
    with torch.no_grad():
        z = torch.randn(n, self.latent_dim)
        x_gen = self.decoder(z)
    return x_gen
```

Variational Autoencoders
(VAEs)

$$\mathscr{L}_{\text{VAE}} = \|x - D_\theta(z)\|^2 + \beta\, D_{KL}(p_\phi(z\,|\,x)\|\ \mathcal{N}(0,1))$$

In summary

1. Two arbitrary networks E_ϕ, D_θ
2. Tractable latent space of arbitrary dimensionality $p(z)$
3. Single-shot sampling

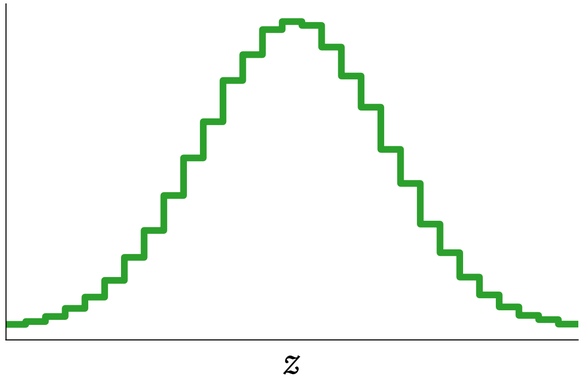
What's the problem?

1. Good on average, bad in details
2. Limited by parametric form of latent space
3. No direct access to likelihood

Learning a transformation

Latent space

$z \sim p(z)$

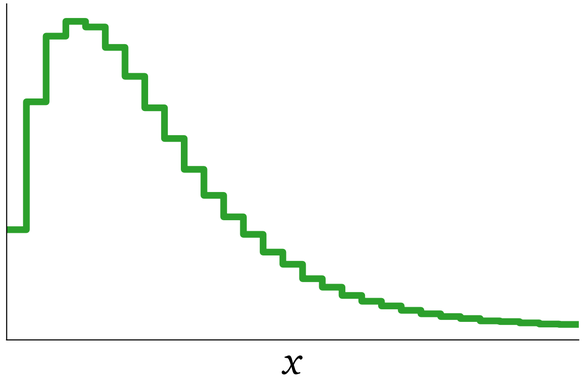


?

A function parametrised by a neural network?

Phase space

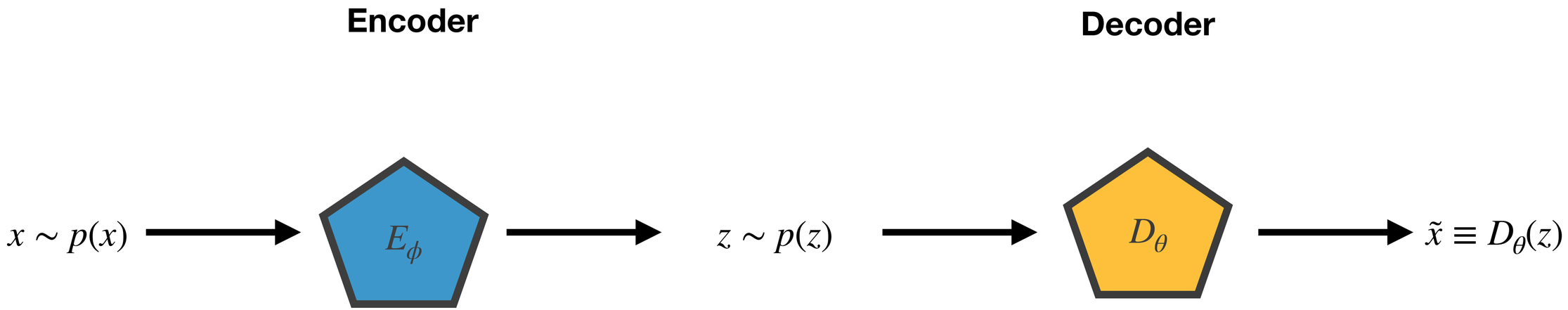
$x \sim p(x)$



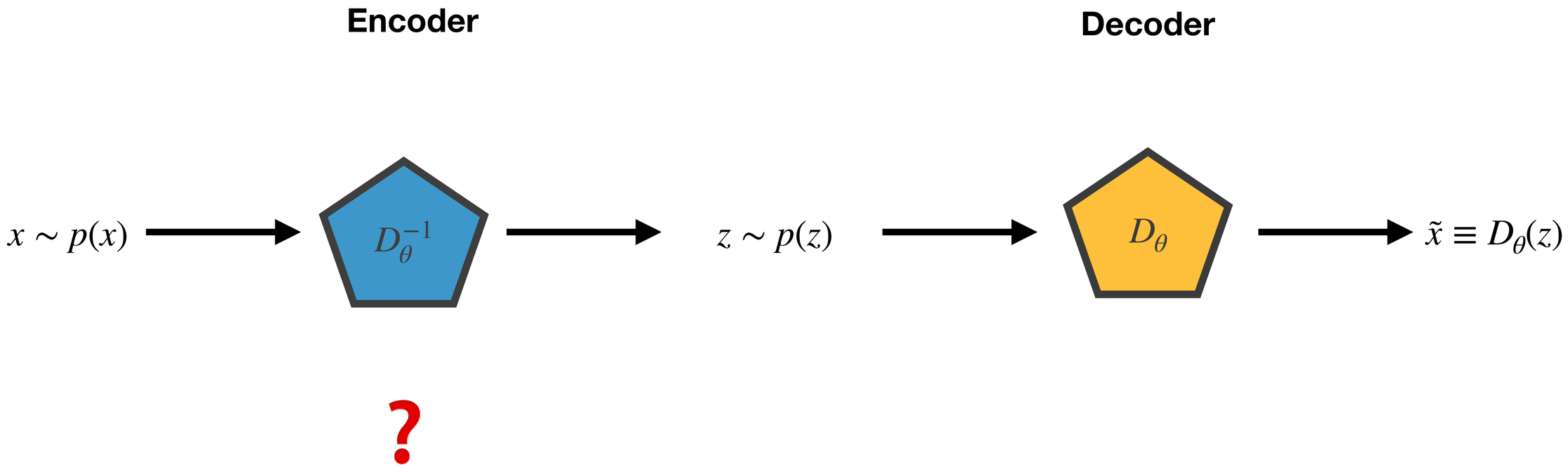
- easy to sample from
- (sometimes) known analytical form
- usually not physically interpretable

- hard to sample from
- (usually) unknown analytical form
- physically interpretable

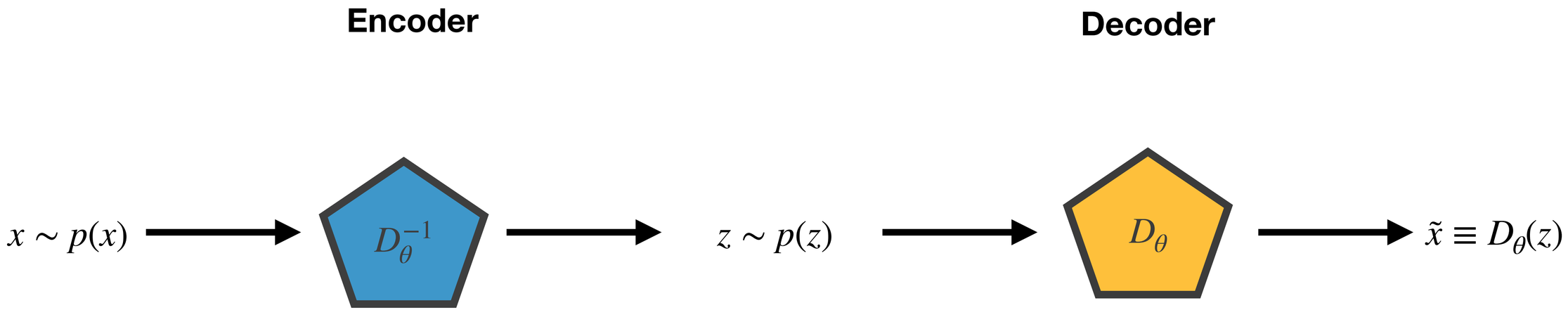
Invertible Neural Networks
(INNs)



Invertible Neural Networks
(INNs)

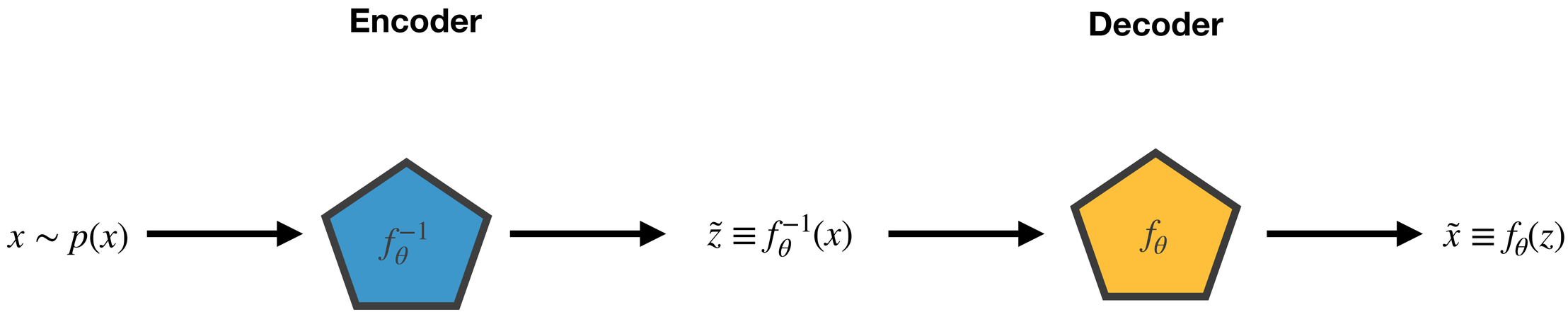


Invertible Neural Networks
(INNs)



Generally, not invertible

Invertible Neural Networks
(INNs)



$$f_\theta(z) \equiv f(z, D_\theta)$$

Invertible Neural Networks
(INNs)

$$f_\theta(z) = x \qquad f_\theta^{-1}(x) = z$$

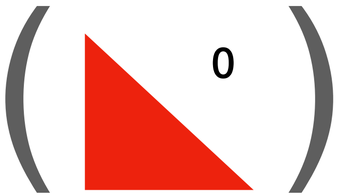
$$p_\theta(x)dx = p(z)dz$$

$$dx = \left| \det \frac{\partial f_\theta}{\partial z} \right| dz$$

$$p_\theta(x) = p(z) \left| \det \frac{\partial f_\theta}{\partial z} \right|^{-1} = p(f_\theta^{-1}(x)) \left| \det \frac{\partial f_\theta^{-1}}{\partial x} \right|$$

What do we want?

1. f_θ needs to be invertible (bijective)
2. The Jacobian determinant must be tractable to compute



Lower triangle matrix

Invertible Neural Networks
(INNs)

What do we want?

1. f_θ needs to be invertible (bijective)

2. The Jacobian determinant must be tractable to compute

How to construct f_θ

1. Autoregressive flows (IAF, MAF)

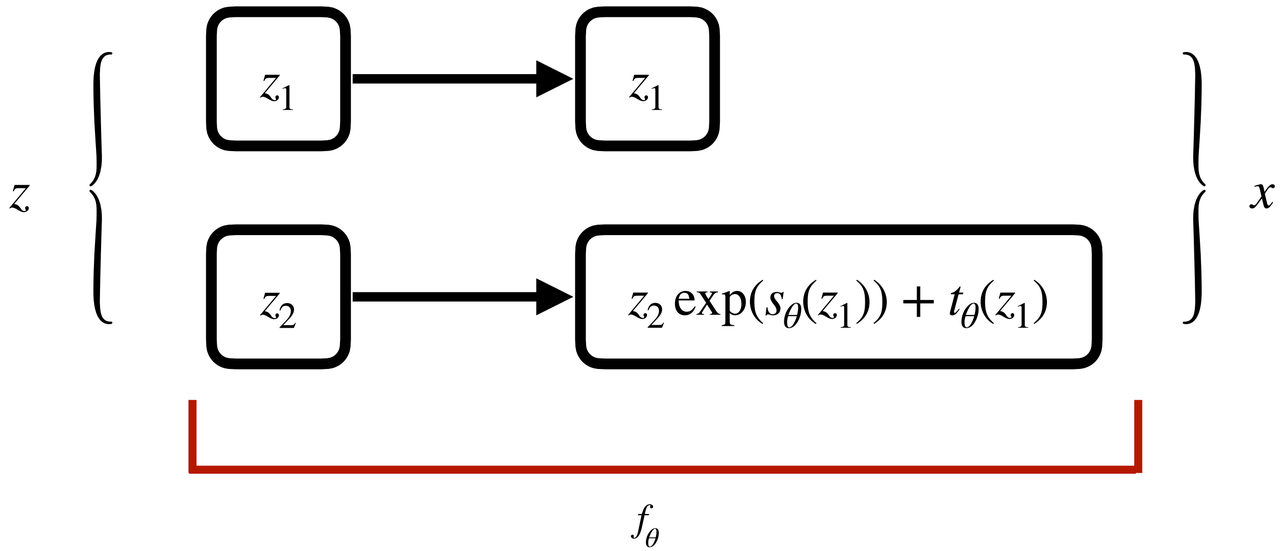
2. Continuous normalising flows

3. Coupling blocks

Invertible Neural Networks
(INNs)

Coupling blocks — Affine Layers

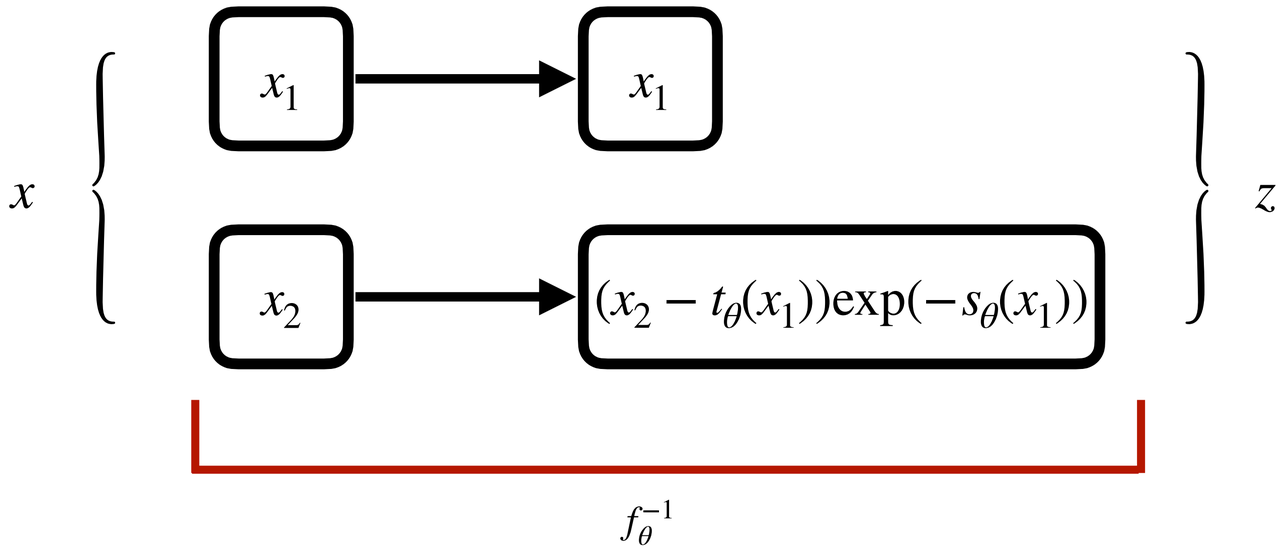
$$D_\theta \equiv s_\theta, t_\theta$$



Invertible Neural Networks
(INNs)

Coupling blocks — Affine Layers

$$D_\theta \equiv s_\theta, t_\theta$$

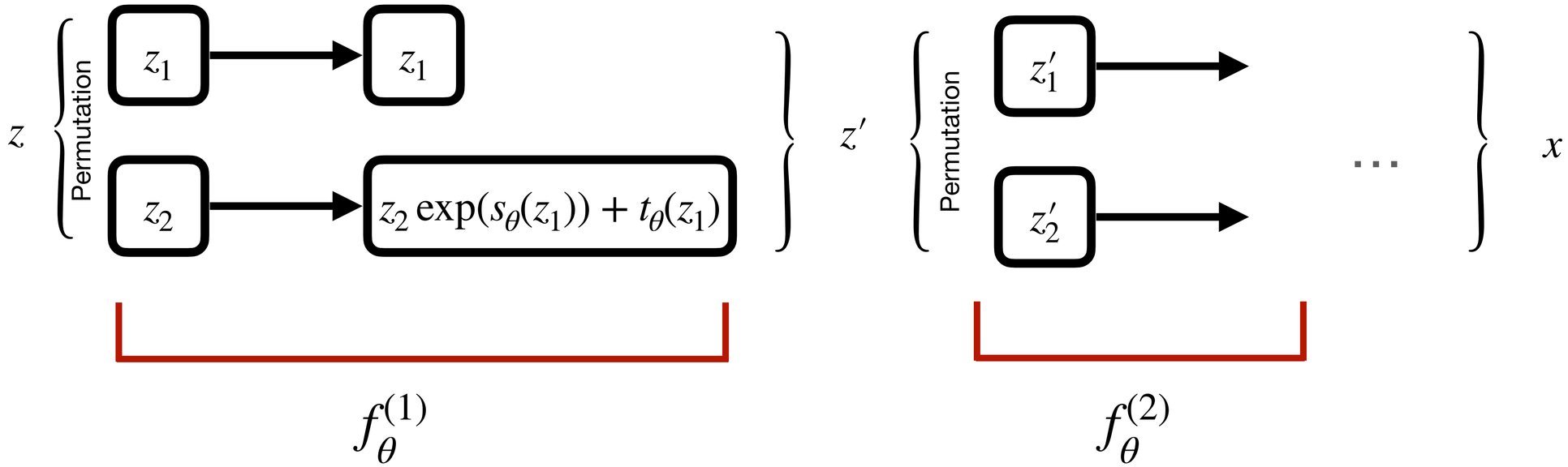


$$\frac{\partial f_\theta^{-1}}{\partial x} = \begin{pmatrix} \frac{\partial f_{\theta,1}^{-1}}{\partial x_1} & \frac{\partial f_{\theta,1}^{-1}}{\partial x_2} \\ \frac{\partial f_{\theta,2}^{-1}}{\partial x_1} & \frac{\partial f_{\theta,2}^{-1}}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \text{finite} & \text{diag}(\exp(-s_\theta(x_1))) \end{pmatrix} \quad \longrightarrow \quad \left| \det \frac{\partial f_\theta^{-1}}{\partial x} \right| = \prod \exp(-s_\theta(x_1))$$

Invertible Neural Networks
(INNs)

Coupling blocks — Affine Layers

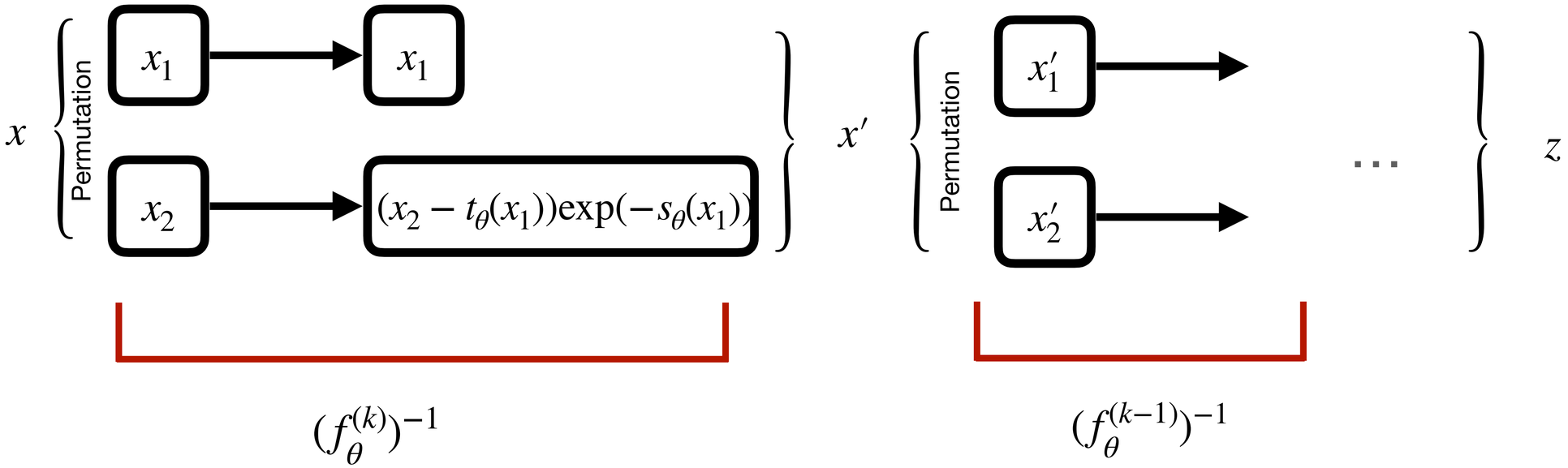
$$f_\theta = f_\theta^{(k)} \circ f_\theta^{(k-1)} \circ \ldots \circ f_\theta^{(2)} \circ f_\theta^{(1)}$$



Invertible Neural Networks
(INNs)

Coupling blocks — Affine Layers

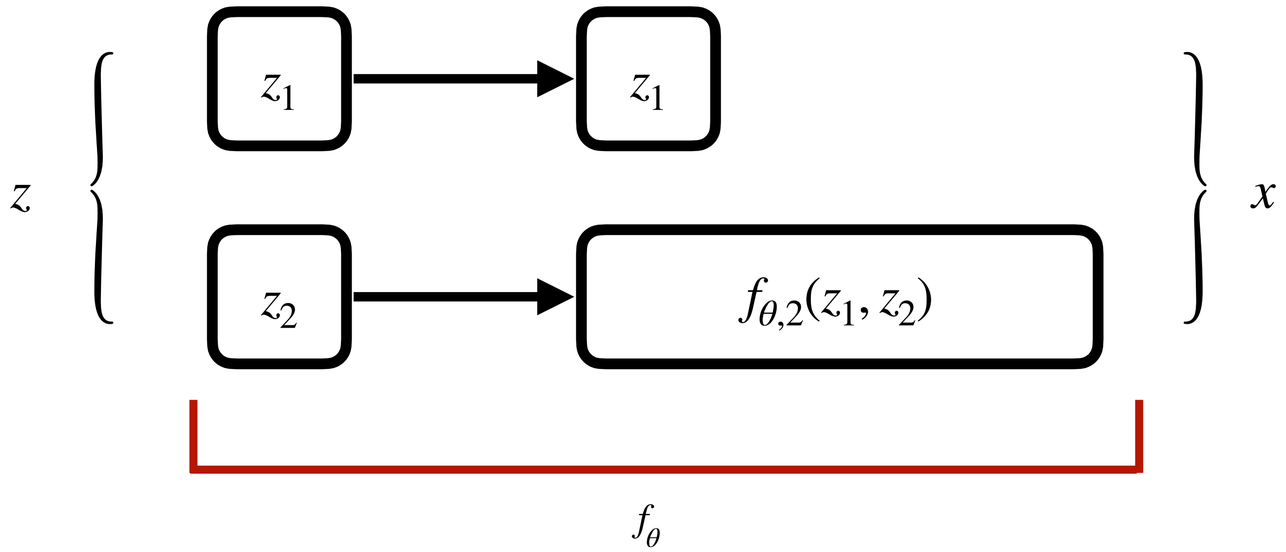
$$f_\theta^{-1} = (f_\theta^{(1)})^{-1} \circ (f_\theta^{(2)})^{-1} \circ \ldots \circ (f_\theta^{(k-1)})^{-1} \circ (f_\theta^{(k)})^{-1}$$



Invertible Neural Networks
(INNs)

Coupling blocks

$D_\theta \equiv \text{tuple(parameters)}$



$f_{\theta,2}(z_1, z_2)$ can be arbitrary as long as invertible in z_2

Spline parametrization (Quadratic Splines, Cubic Splines, Rational Quadratic Splines, etc)

Training

```python
def batch_loss(self, x):
    """
    Args:
        x: real samples from phase space, shape (batch_size, ...)

    Returns:
        loss: -log p(x), averaged over batch
    """
    # Inverse pass: x -> z, accumulating log|det df^-1/dx| across blocks
    z, log_det = self.block_transformation(x, inverse=True)

    # log p(x) = log p(z) + log |det df^-1/dx|
    log_p_x = self.latent.log_prob(z).sum(dim=1) + log_det

    return -log_p_x.mean()
```

Sampling

```
def sample(self, n):
    z = self.latent.sample((n,))
    x, _ = self.block_transformation(z, inverse=False)
    return x
```

Invertible Neural Networks
(INNs)

$$\mathscr{L}_{\text{INN}} = -\log p_\theta(x)$$

In summary

1. One arbitrary networks D_θ
2. Tractable latent space of fixed dimensionality $p(z)$
3. Single-shot sampling
4. Fast access to likelihood

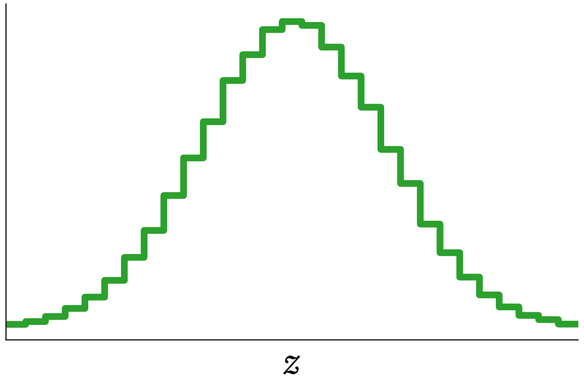
What's the problem?

1. Invertibility constraints
2. Latent space constraints
3. Can be slow in inference or training

Learning a transformation

Latent space

$z \sim p(z)$

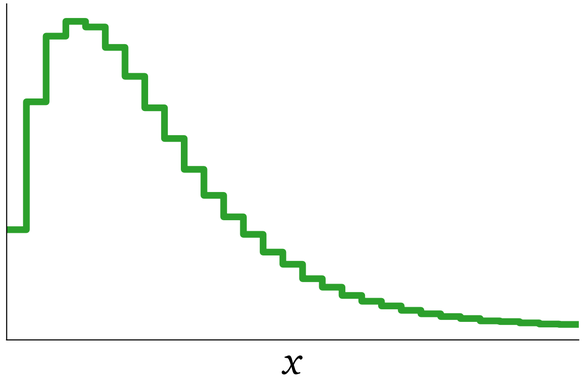


?

A stochastic process parametrised by a neural network?

Phase space

$x \sim p(x)$



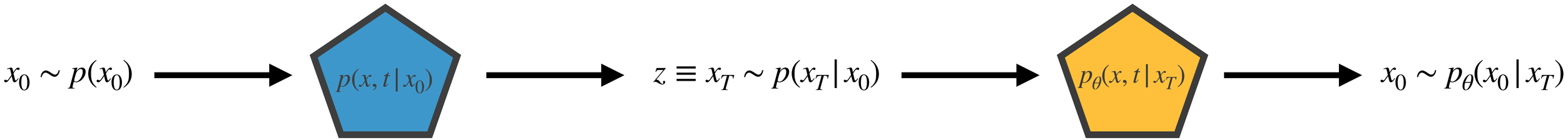
- easy to sample from
- (sometimes) known analytical form
- usually not physically interpretable

- hard to sample from
- (usually) unknown analytical form
- physically interpretable

Diffusion Networks

Forward (Diffusion process)

Backwards (Denoising process)



Diffusion Networks

Forward (Diffusion process)

SDE: $dx = f(x,t)dt + g(t)\,dW$

Drift Diffusion

Backwards (Denoising process)

SDE: $dx = \left(f(x,t) - g(t)^2 \nabla_x \log p(x,t)\right) dt + g(t)\,d\bar{W}$

Score

Choose $f(x,t)$ s.t. forward process becomes tractable

$$f(x,t) = xa(t) \longrightarrow p(x,t\,|\,x_0) = \mathcal{N}(\mu(t),\sigma(t)) \longrightarrow x(t) = \mu(t) + \epsilon\sigma(t) \quad \text{with} \quad \epsilon \sim \mathcal{N}(0,1)$$

Need to make sure that $p(x,t=0) = p(x_0)$ and $p(x,t=T) = \mathcal{N}(0,1)$

Forward model completely determined, but backwards?

Diffusion Networks

Forward (Diffusion process)

SDE: $dx = f(x,t)dt + g(t)\ dW$

Backwards (Denoising process)

SDE: $dx = \left(f(x,t) - g(t)^2 \nabla_x \log p(x,t)\right) dt + g(t)\ d\bar{W}$

Forward model completely determined, but backwards?

Unknown part is the score function

$$\|s_\theta(x,t) - \nabla_x \log p(x,t)\|^2$$

Diffusion Networks

Forward (Diffusion process)

SDE: $dx = f(x,t)dt + g(t)\ dW$

Backwards (Denoising process)

SDE: $dx = \left(f(x,t) - g(t)^2 \nabla_x \log p(x,t)\right) dt + g(t)\ d\bar{W}$

Forward model completely determined, but backwards?

Unknown part is the score function

$$\arg\min_\theta \|s_\theta(x,t) - \nabla_x \log p(x,t)\|^2 = \arg\min_\theta \|s_\theta(x,t) - \nabla_x \log p(x,t\,|\,x_0)\|^2 = \arg\min_\theta \|s_\theta(x,t) + \frac{\epsilon}{\sigma(t)}\|^2 = \arg\min_\theta \|\epsilon_\theta(x,t) - \epsilon\|^2$$

Score-matching

DDPM

Diffusion Networks

Forward (Diffusion process)

SDE: $dx = f(x,t)dt + g(t)\ dW$

Backwards (Denoising process)

SDE: $dx = \left(f(x,t) - g(t)^2 \nabla_x \log p(x,t)\right) dt + g(t)\ d\bar{W}$

Once, score is known

$$\int_{x_0}^{x_T} dx = \int_0^T dt\ \left(f(x,t) - g(t)^2 \nabla_x \log p(x,t)\right) + \int_0^T d\bar{W}(t)\ g(t)$$

$$x_0 = x_T - \int_0^T \left(f(x,t) - g(t)^2 \nabla_x \log p(x,t)\right) - \int_0^T d\bar{W}(t)\ g(t)$$

Diffusion Networks

Design choices

1. Exact choice of loss
2. t- sampling or SDE solver (discrete grid, continuous interval)
3. Choosing $\mu(t)$ and $\sigma(t)$

Example DDPMs

1. Denoising-loss $\mathscr{L}_{\text{DDPM}} = \|\epsilon_\theta(x, t) - \epsilon\|^2$
2. Discrete t-sampling with an Euler-Maruyama solver
3. $\mu_t = \sqrt{1 - \bar{\beta}_t} x_0 \quad \text{and} \quad \sigma_t = \sqrt{\bar{\beta}_t} \qquad \text{with} \qquad 1 - \bar{\beta}_t = \prod_{i=0}^{t} (1 - \beta_i) \quad \text{and} \quad \beta_i$ noise-scheduler

Denoising Diffusion Probabilistic Models (DDPMs)

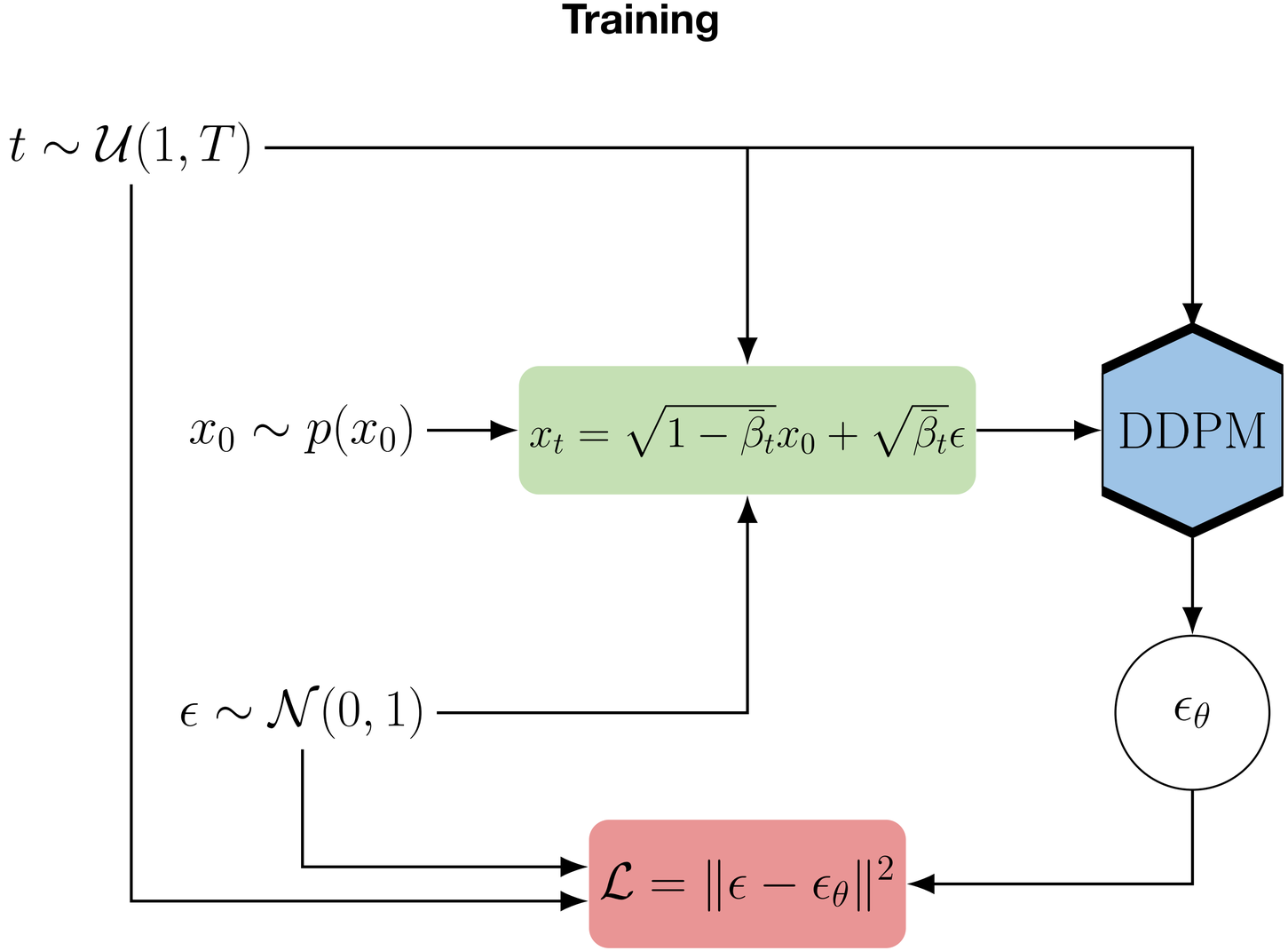


Figure from arXiv:2305.10475

Denoising Diffusion Probabilistic Models
(DDPMs)

Sampling

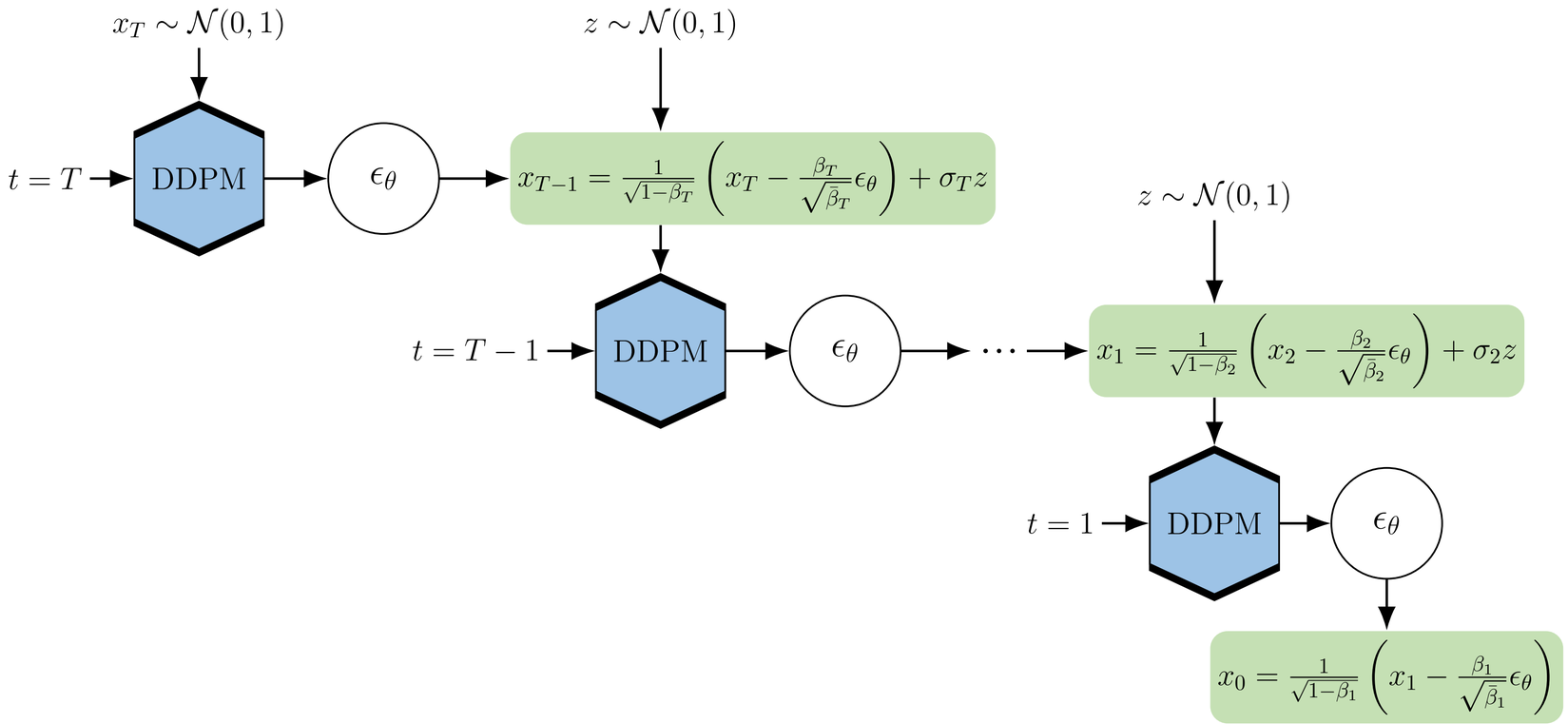


Figure from arXiv:2305.10475

Denoising Diffusion Probabilistic Models
(DDPMs)

$$\mathscr{L}_{\text{DDPM}} = \|\epsilon_\theta(x, t) - \epsilon\|^2$$

In summary

1. One arbitrary networks ϵ_θ
2. Gaussian latent space of fixed dimensionality $p(x_T)$
3. Multi-shot sampling (T times)
4. Access to likelihood

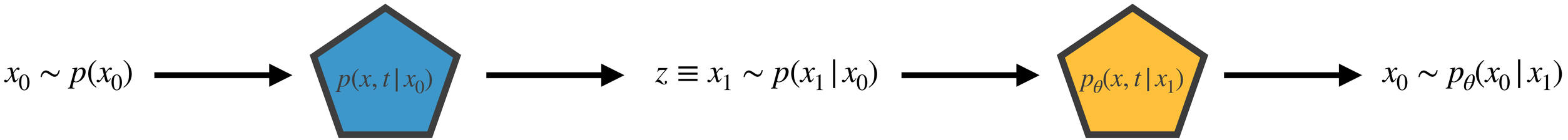
What's the problem?

1. Latent space constraints to be Gaussian
2. Slow inference
3. Likelihood even slower

Conditional Flow Matching
(CFM)

Forward (Diffusion process)

Backwards (Denoising process)



Conditional Flow Matching
(CFM)

	Forward (Diffusion process)	**Backwards** (Denoising process)
Diffusion	SDE: $dx = f(x,t)dt + g(t)\ dW$	SDE: $dx = \left(f(x,t) - g(t)^2 \nabla_x \log p(x,t)\right) dt + g(t)\ d\bar{W}$
CFM	ODE: $dx = \underbrace{\left(f(x,t) - \frac{1}{2} g(t)^2 \nabla_x \log p(x,t)\right)}_{\text{Velocity } v(x,t)} dt$	ODE: $dx = -\left(f(x,t) - \frac{1}{2} g(t)^2 \nabla_x \log p(x,t)\right) dt$

In this setup: Once you know $v(x,t)$ → 💸

What's different?

1. Drop all randomness
2. Time-symmetric
3. Instead of learning score, learning velocity field directly

Conditional Flow Matching
(CFM)

ODE: $dx = v(x, t)dt$

How can we learn the velocity?

$$||v_\theta(x, t) - v(x, t)||^2$$

Conditional Flow Matching
(CFM)

ODE: $dx = v(x,t)dt$

How can we learn the velocity?

$$\arg\min_{\theta} \|v_\theta(x,t) - v(x,t)\|^2 = \arg\min_{\theta} \|v_\theta(x,t) - v(x,t\,|\,x_0)\|^2$$

Construct a tractable conditional velocity field through simple trajectories

Example: **Linear trajectories**

$$x(t) = (1-t)x_0 + tx_1$$

$$v(x,t\,|\,x_0) = \frac{dx}{dt} = x_1 - x_0$$

Conditional Flow Matching
(CFM)

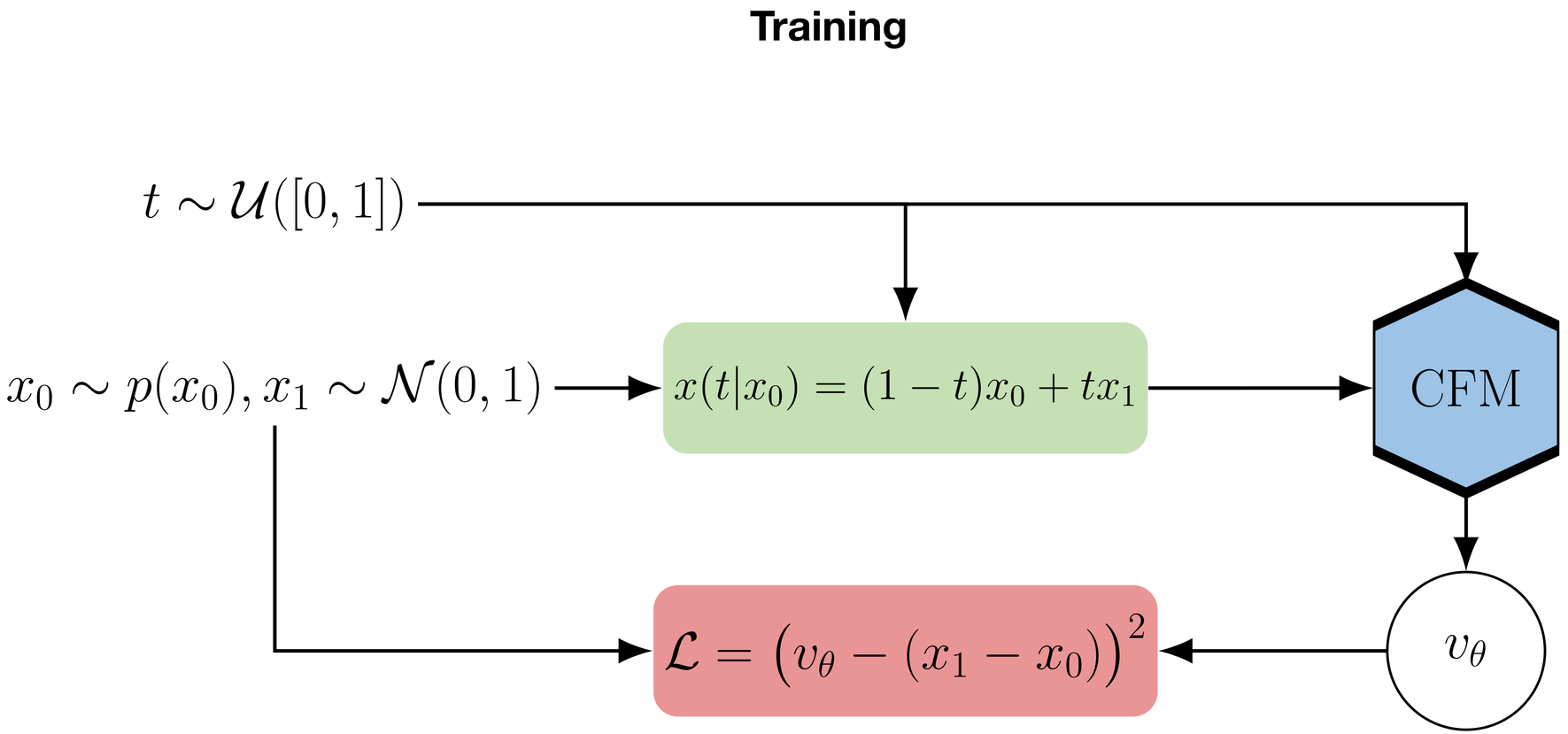


Figure from arXiv:2305.10475

Conditional Flow Matching
(CFM)

Sampling

$$x_0 = x_1 - \int_0^1 dt$$

Conditional Flow Matching
(CFM)

$$\mathscr{L}_{\text{CFM}} = \|v_\theta(x,t) - v(x,t\,|\,x_0)\|^2$$

In summary

1. One arbitrary networks v_θ
2. Arbitrary latent space of fixed dimensionality $p(x_1)$
3. Multi-shot sampling
4. Access to likelihood

What's the problem?

1. Slow inference
2. Likelihood even slower

Autoregressive Networks

n-dimensional phase space with $p(x_1, \ldots, x_n) = \prod_i p(x_i \,|\, x_{<i})$

Most famous example: GPT-style LLMs

$$p(\text{ML is cool.}) = p(\text{ML})\; p(\text{is} \,|\, \text{ML})\; p(\text{cool} \,|\, \text{ML, is})\; p(\text{ . } |\, \text{ML, is, cool})$$

What's the difference?

1. Fitting 1d distribution is easy, multi-dimensional ones are hard
2. Factor n-dimensional joint into n 1-dimensional conditionals (chain rule)
3. No latent space — directly model $p(x)$

Autoregressive Networks

n-dimensional phase space with $p(x_1, \ldots, x_n) = \prod_i p(x_i \,|\, x_{<i})$

Most famous example: GPT-style LLMs

$$p(\text{ML is cool.}) = p(\text{ML})\; p(\text{is} \,|\, \text{ML})\; p(\text{cool} \,|\, \text{ML, is})\; p(\,. \,|\, \text{ML, is, cool})$$

How to train?

$$-\log p_\theta(x_1, \ldots, x_n) = -\sum_i \log p_\theta(x_i \,|\, x_{<i})$$

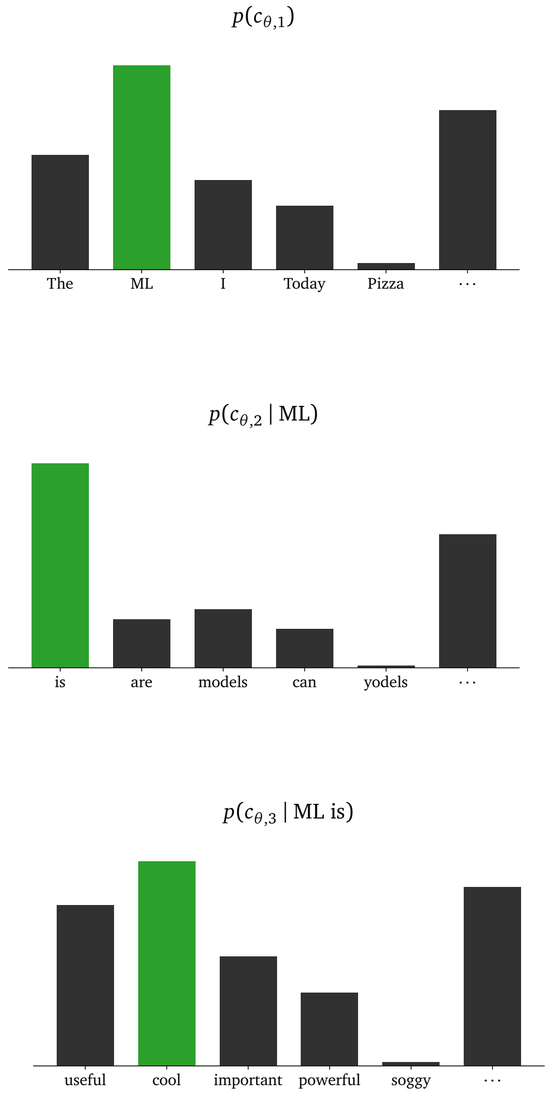
Autoregressive Networks

Design choices:

1. 1d parametrisation (binned, Gaussian mixture etc.)
2. Autoregressive order (natural order?)

Example: LLMs

1. Categorical distribution over entire vocabulary
   (Learned parameters are bin-heights)
2. Language has natural beginning and end points



Autoregressive Networks

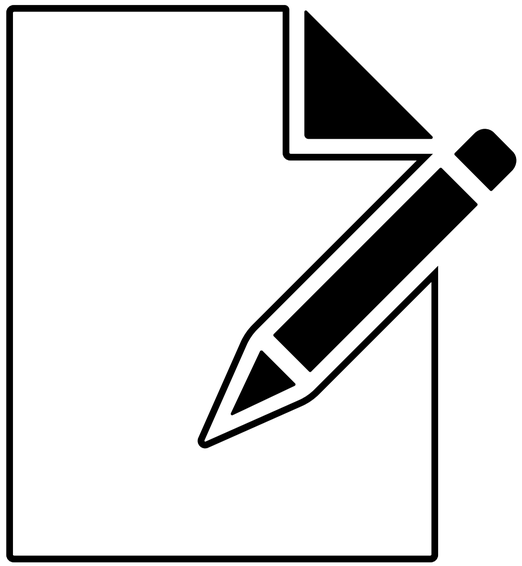
$$\mathscr{L}_{\text{AR}} = -\log p_\theta(x)$$

In summary

1. One arbitrary networks c_θ
2. No latent space
3. Autoregressive sampling
4. Access to likelihood

What's the problem?

1. Slow inference
2. Limited by parametric form
3. Autoregressive order not always trivial

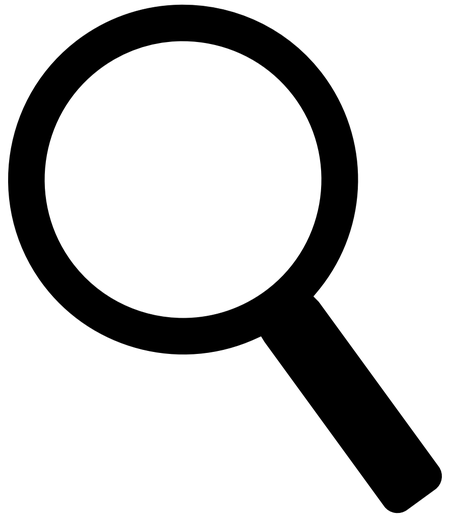


2. Finding the right problem formulation

Generative networks in HEP are used to …

1. Fast simulation through surrogate models (calorimeter showers, parton showers, End-to-End ,…)

2. Density estimation for
    - Anomaly Detection
      (background estimation)
    - SBI (NPE, NLE)
      (parameter inference)
    - Unfolding
      (correcting for detector effects)
    - Neural importance sampling
      (learning a proposal function)
    - Improve hadronization
      (learning data density)
    - Superresolution
    - …



3. Validating generated samples

Validation
Classifier metric

What we have?

Samples $x \sim p_{\text{model}}(x)$ and $x \sim p_{\text{data}}(x)$

What do we want to know?

Is $p_{\text{model}}(x) = p_{\text{data}}(x)$?

How to test?

Classifier train on samples from $p_{\text{model}}(x)$ (class 0) and $p_{\text{data}}(x)$ (class 1)

Why?

CONVERGED classifier output $C_\theta(x)$ can be translated to

$$w(x) = \frac{p_{\text{data}}(x)}{p_{\text{model}}(x)} \approx \frac{C_\theta(x)}{1 - C_\theta(x)}$$

Which model did best?

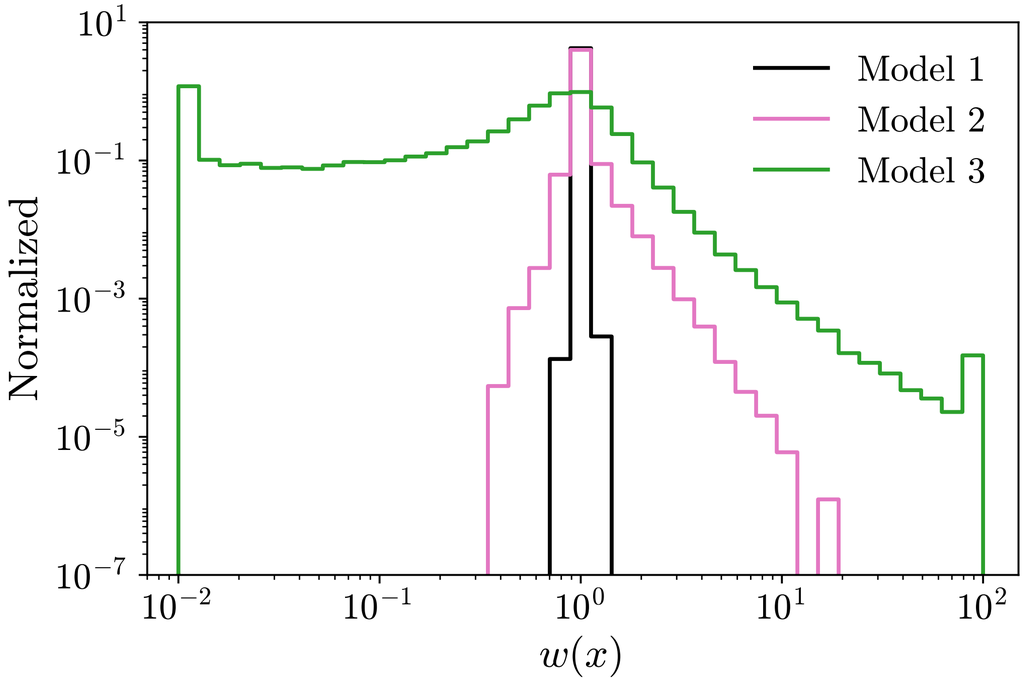


Figure from arXiv:2605.30453

Validation

Classifier metric

Which model did best?

Model 1: Sharp peak around $w(x) = 1$, no tails

→ Essentially $p_{\text{model}}(x) = p_{\text{data}}(x)$, no localised mismodelling

Model 2: Peak around $w(x) = 1$, tails towards larger weights

→ In bulk $p_{\text{model}}(x) = p_{\text{data}}(x)$, model underpopulates certain phase space region

Model 3: Washed out peak at $w(x) = 1$, large tails on both sides

→ model underpopulates certain phase space regions (large weights), overpopulates others (small weights)

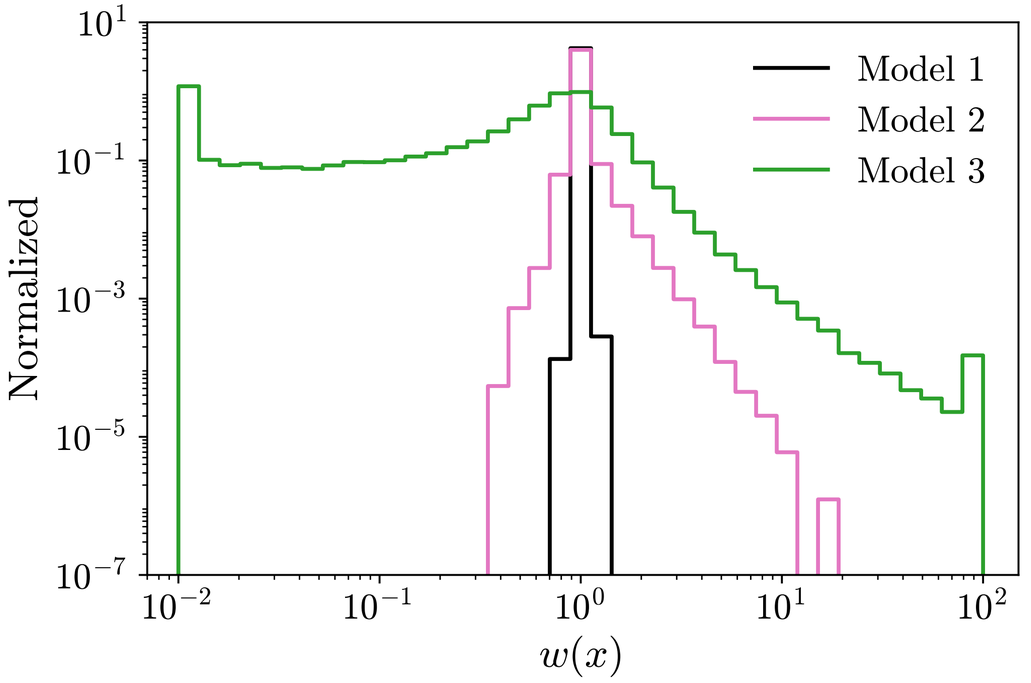


Figure from arXiv:2605.30453

What you should take away from this lecture

People have thought about a lot of **different ways** to **formulate generative models**

Generative algorithms **are easy to understand** (and often times even **physics inspired**)

There’s not **THE generative network** that is s.o.a in every task

Generative models can be used **extremely versatile** in HEP and beyond

It’s always good to check whether generative networks did **good** & whether they are actually **solving a given task**

References

General:
Tilman Plehn, Anja Butter, Barry Dillon, Theo Heimel, Claudius Krause Ramon Winterhalder; *Modern Machine Learning for LHC Physicists*, arXiv:2211.01421

Sascha Diefenbacher, SPS and Gregor Kasieczka; *Generative Models and Statistical Validation*, arXiv:2605.30453

GAN: I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, Generative adversarial networks, arXiv:1406.2661

VAE: D. P. Kingma and M. Welling, Auto-encoding variational bayes, arXiv:1312.6114

INN: G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, and B. Lakshminarayanan, Normalizing flows for probabilistic modeling and inference, arXiv:1912.02762

Diffusion:
Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole, Score-based generative modeling through stochastic differential equations, arXiv:2011.13456

J. Ho, A. Jain, and P. Abbeel, Denoising diffusion probabilistic models, arXiv:2006.11239

CFM: Y. Lipman, R. T. Q. Chen, H. Ben-Hamu, M. Nickel, and M. Le, Flow matching for generative modeling, arXiv:2210.02747

Classifier Metric:
G. Grosso, M. Letizia, M. Pierini, and A. Wulzer, Goodness of fit by Neyman-Pearson testing, SciPost Phys. 16 (2024) 5, 123, arXiv:2305.14137

R. Das, L. Favaro, T. Heimel, C. Krause, T. Plehn, and D. Shih, How to understand limitations of generative networks, SciPost Phys. 16 (2024) 1, 031, arXiv:2305.16774

Amplification:
A. Butter, S. Diefenbacher, G. Kasieczka, B. Nachman, and T. Plehn, GANplifying event samples, SciPost Phys. 10 (2021) 6, 139, arXiv:2008.06545

H. Bahl, S. Diefenbacher, N. Elmer, T. Plehn, and J. Spinner, Forecasting Generative Amplification, arXiv:2509.08048