Deep Generative Models

Chapter 3: Generation by Explicit Distribution Learning

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering

University of Toronto

Summer 2025

Latent Space: *Motivation*

As we have seen several times: *although data samples are high-dimensional, their valid cases are significantly limited within the data space*

samples might be thought as if processed from a much easier origin

+ *What do you mean by an easy origin?*
– *We have seen such things a lot! An example makes life easier*

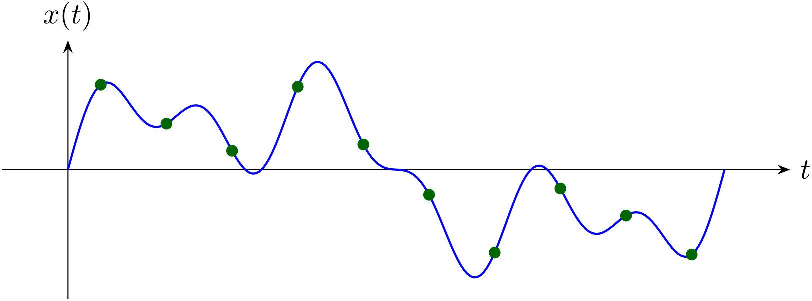
Example: *We listen to a set of audio signals. We know that these signals have at most* m *known harmonics*, i.e.,

$$x(t) = \sum_{j=1}^{m} z_j \sin(2\pi f_j t)$$

where z_j *is the amplitude of harmonic* j*. We want to learn data distribution.*

Latent Representation: *Example*

To make data samples, we could work directly in time domain: we sample $d \gg 1$ time samples according to Nyquist theorem

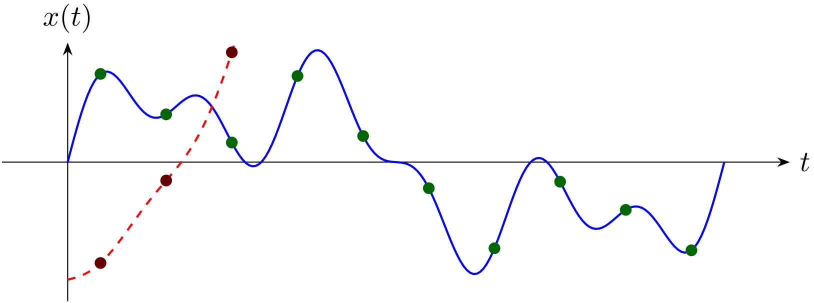


So, data-space in this case is $\mathbb{R}^d$ with d being potentially very large!

$$x = [x_1, x_2, \dots, x_d]$$

Latent Representation: *Example*

We however know that many combinations of samples are impossible!



Examples: *Though data-space is $\mathbb{R}^d$, samples of exponential-like signals are not happening ⇝ invalid data samples!*

Latent Representation: *Example*

An alternative way of describing data distribution is to learn the distribution of harmonic amplitudes, i.e.,

$$z = [z_1, \ldots, z_m]$$

If we know $P(z)$: we can sample $z \sim P(z)$ and generate an audio signal as

$$x(t) = \sum_{j=1}^{m} z_j \sin(2\pi f_j t)$$

This gives a very simple example of what is known as

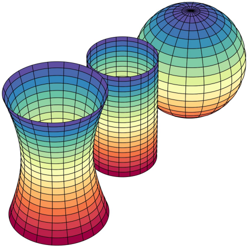
latent representation of data

Data Manifold

Valid data is typically concentrated in a

narrow (low-dimensional) manifold hidden in the data-space

+ *What do you mean by a manifold?!*
- Think of a spherical surface: *it looks 3D, but we can only move 2D when we are on it ⇝ it's a 2D manifold embedded in 3D space*



Valid data-points are also the same

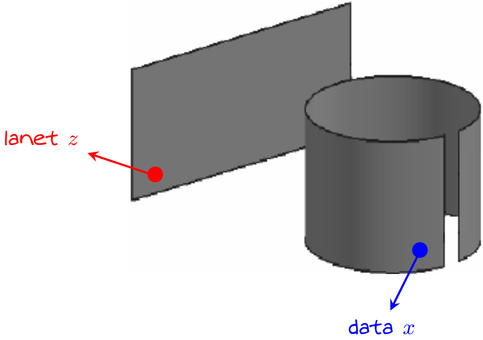
↳ *They are high-dimensional*

↳ *They lie on a thin manifold*

Latent Space

Latent Space

Latent space can be thought of as a coordinate system for (or a transformed version of) data manifold, potentially in lower dimensions



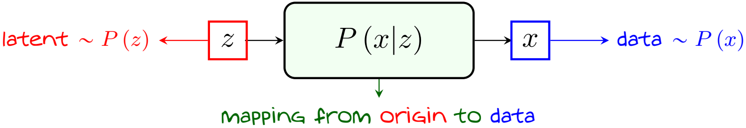
Working with *Latent Representation*

Considering the notion of latent space: *in most cases, it's more efficient to work with latent representations of data*

- *Many times, it's easier to learn the distribution of latent representation*
  - ↳ *In audio example: assuming time samples in x being i.i.d. is completely non-sense; however, having i.i.d. latent z could make sense!*
  - ↳ *Even if $\hat{P}(z)$ is not accurate, a poor sample z can still lead to a sensible audio signal*
- *Typically, latent representation is much lower in dimensions than data*
  - ↳ *MNIST images could have as small as 8D latent space*
- *But, we could have latent representation of the same dimension*
  - ↳ *In this case, the latent space deforms the complicated data manifold into a simpler manifold in the same space*
    - ↳ *e.g., a complicated 2D surface in 3D is transformed to a simple 2D plane in 3D*

Latent Representation: *Statistical Interpretation*

From statistical viewpoint: *we can look at the latent representation as the root which has been processed to a visible form of data*



In this formulation, we can say

$$P(z,x) = P(x|z)\, P(z) \rightsquigarrow P(x) = \sum_z P(z,x) = \sum_z P(x|z)\, P(z)$$

Attention

There is no unique latent space: we have various (potentially infinite) choices for latent space $\rightsquigarrow$ *each choice has its own* $P(z)$ *and mapping* $P(x|z)$

Normalizing Flow: *A Simple Probability Problem*

Assume that *we have the continuous random variable* z

$$z \sim Q(z)$$

We pass it through function $f : \mathbb{R} \mapsto \mathbb{R}$ *and compute* x

$$x = f(z)$$

with f *having the following properties*

- f is invertible, i.e., we can find $z = f^{-1}(x)$
- f is strictly increasing, i.e., $z_1 < z_2 \rightsquigarrow f(z_1) < f(z_2)$
  - ↳ *This concludes that* f^{-1} *is also strictly increasing*

We want to find the density of x

Change of Variable

Let's start with computing the cumulative distribution of x

$$\begin{aligned}\mathrm{CDF}_x(a) = \Pr\{x \leqslant a\} &= \Pr\{f(z) \leqslant a\}\\ &= \Pr\left\{z \leqslant f^{-1}(a)\right\} = \mathrm{CDF}_z\left(f^{-1}(a)\right)\end{aligned}$$

By definition, the distribution of x is the derivative of CDF, i.e.,

$$\begin{aligned}\text{density of } x \text{ at } a \equiv P(a) &= \frac{\mathrm{d}}{\mathrm{d}a}\mathrm{CDF}_x(a)\\ &= \frac{\mathrm{d}}{\mathrm{d}a}\mathrm{CDF}_z\left(f^{-1}(a)\right)\\ &= \frac{\mathrm{d}}{\mathrm{d}u}\mathrm{CDF}_z(u)\,|_{u=f^{-1}(a)}\frac{\mathrm{d}}{\mathrm{d}a}f^{-1}(a)\\ &= Q\left(f^{-1}(a)\right)\frac{\mathrm{d}}{\mathrm{d}a}f^{-1}(a)\end{aligned}$$

Change of Variable

Replacing a with x for simplicity, we have

$$P(x) = Q\left(f^{-1}(x)\right)\frac{\mathrm{d}}{\mathrm{d}x}f^{-1}(x)$$

As we know f is invertible, we can write

$$z = f^{-1}(x) \leftrightsquigarrow f(z) = x$$

Taking derivative, we have

$$\left.\begin{aligned} \mathrm{d}z &= \frac{\mathrm{d}}{\mathrm{d}x}f^{-1}(x)\,\mathrm{d}x \\ \dot{f}(z)\,\mathrm{d}z &= \mathrm{d}x \end{aligned}\right\} \rightsquigarrow \frac{\mathrm{d}}{\mathrm{d}x}f^{-1}(x) = \frac{1}{\dot{f}(z)} = \frac{1}{\dot{f}\left(f^{-1}(x)\right)}$$

Change of Variable: *Scalar Result*

It is easy to show that

- *Having an increasing f is not needed*
  - ↳ *For decreasing f, everything holds with a sign change*
- *It's though necessary for f to be invertible*

Change of Variable *(Scalar)*

If we pass $z \sim Q(z)$ through an invertible transform $x = f(z)$; then,

$$x \sim P(x) = \frac{Q\left(f^{-1}(x)\right)}{\left|\dot{f}\left(f^{-1}(x)\right)\right|}$$

Change of Variable: *Array Variables*

Change of Variable *(Vectorized)*

Let $z \in \mathbb{R}^d$ be distributed by $z \sim Q(z)$ and $f : \mathbb{R}^d \mapsto \mathbb{R}^d$ be invertible and differentiable. Then, the transform $x = f(z)$ is distributed by

$$x \sim P(x) = \frac{Q(z)}{\det |\nabla f(z)|} \quad \textit{with} \quad z = f^{-1}(x)$$

1. ∇f *is a $d \times d$ Jacobian matrix*

$$\nabla f(z) = \begin{bmatrix} \mathrm{d}x_1/\mathrm{d}z_1 & \cdots & \mathrm{d}x_1/\mathrm{d}z_d \\ \vdots & \ddots & \vdots \\ \mathrm{d}x_d/\mathrm{d}z_1 & \cdots & \mathrm{d}x_d/\mathrm{d}z_d \end{bmatrix}$$

Change of Variable: *Array Variables*

Change of Variable *(Vectorized)*

Let $z \in \mathbb{R}^d$ be distributed by $z \sim Q(z)$ and $f : \mathbb{R}^d \mapsto \mathbb{R}^d$ be invertible and differentiable. Then, the transform $x = f(z)$ is distributed by

$$x \sim P(x) = \frac{Q(z)}{\det |\nabla f(z)|} \quad \textit{with} \quad z = f^{-1}(x)$$

2. *f should be differentiable to ∇f exists and invertible to*

$$\det|\nabla f(z)| \neq 0$$

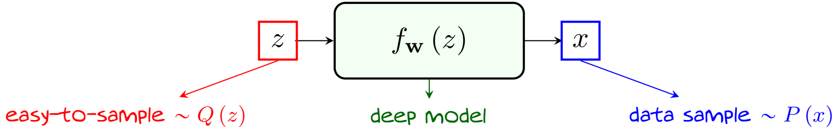
3. *To have an invertible transform, z and x should be of the same dimension!*

Flow: *Learnable Mapping from Latent to Data*

It is hard to learn distribution directly in the data-space

- *It lies close to a very thin manifold hidden in the data-space*
  - ↳ *Data distribution is very complex and the model can hardly fit to it*
- *Even if we learn it* $\rightsquigarrow$ *it's very hard to sample from it!*

\+ *What if we focus on a latent space with simple distribution?*



Assuming mapping to be deterministic and learnable $\rightsquigarrow$ *data distribution is specified in terms of this model using change of variable*

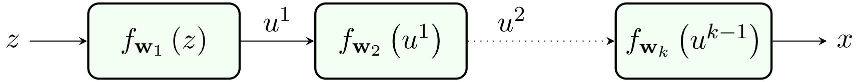
$$P_{\mathbf{w}}(x) = \frac{Q(z)}{\det |\nabla f_{\mathbf{w}}(z)|} \quad \text{with} \quad z = f_{\mathbf{w}}^{-1}(x)$$

Flow Process: *Chain of Mappings*

Typically, we deal with deep models, i.e., *we process features sequentially: if*

- *every transform in the model is invertible and differentiable*

Then, distribution of each feature is given by change of variable



This describes a flow process in which

an easy-to-sample latent representation gradually evolves to a data sample

We can make this happen if we design $f_{\mathbf{w}_1}, \ldots, f_{\mathbf{w}_k}$ such that

the distribution of final output matches the data distribution

Flow Process

Flow Process

Flow process describes sequential evolution of latent z to data sample x as

$$z = u^0 \xrightarrow{f_{\mathbf{w}_1}} u^1 \xrightarrow{f_{\mathbf{w}_2}} \cdots \xrightarrow{f_{\mathbf{w}_{k-1}}} u^{k-1} \xrightarrow{f_{\mathbf{w}_k}} u^k = x$$

We recover latent representation of data sample x by inverse flow as

$$z = u^0 \xleftarrow{f^{-1}_{\mathbf{w}_1}} u^1 \xleftarrow{f^{-1}_{\mathbf{w}_2}} \cdots \xleftarrow{f^{-1}_{\mathbf{w}_{k-1}}} u^{k-1} \xleftarrow{f^{-1}_{\mathbf{w}_k}} u^k = x$$

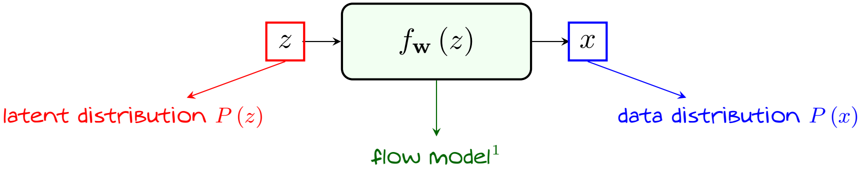
and evaluate distribution of each flow sample u^i using normalizing flow as

$$P_i\left(u^i\right) = \frac{P_{i-1}\left(u^{i-1}\right)}{\det\left|\nabla f_{\mathbf{w}_i}\left(u^{i-1}\right)\right|} \quad \text{with} \quad u_i = f^{-1}_{\mathbf{w}_i}\left(u^i\right)$$

Flow-based Models

Flow-based Model

A flow-based model learns a flow process which evolves a predefined latent distribution $P(z)$ to the data distribution



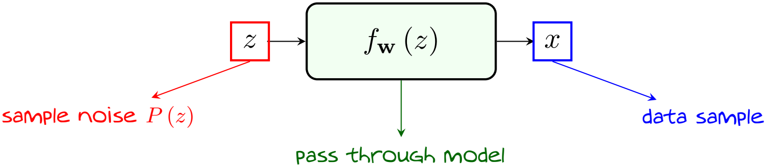
In flow-based models

- *Latent distribution is an easy-to-sample one, e.g., standard Gaussian*
- *Flow model needs to be invertible*[1]

[1] *We represent overall deep model compactly by $f_{\mathbf{w}}$*

Flow-based Models: *Sampling*

Sampling is very easy with flow-based models



`Sample_Flow():`

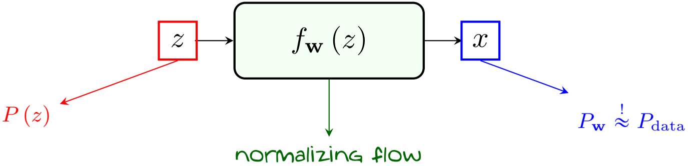
1: *Sample latent as* $z \sim P(z)$ `#Simple noise distribution`

2: *Pass through flow model* $x \leftarrow f_{\mathbf{w}}(z)$

3: **return** x

Flow-based Models: *MLE Learning*

To train: *we can maximize the likelihood of the normalized flow*



For given sample $x \rightsquigarrow$ *compute latent as* $z_{\mathbf{w}} = f_{\mathbf{w}}^{-1}(x)$ *and then likelihood as*

$$P_{\mathbf{w}}(x) = \frac{P(z_{\mathbf{w}})}{\det |\nabla f_{\mathbf{w}}(z_{\mathbf{w}})|}$$

The log-likelihood is hence given by

$$\log P_{\mathbf{w}}(x) = \log P(z_{\mathbf{w}}) - \log \det |\nabla f_{\mathbf{w}}(z_{\mathbf{w}})|$$

Flow-based Models: *MLE Learning*

On a batch of n data samples $\left\{x^j \sim P_{\rm data} : j = 1, \ldots, n\right\}$, we have

$$\begin{aligned}\hat{R}(\mathbf{w}) &= -\frac{1}{n}\sum_j \log P_{\mathbf{w}}(x) \\ &= \frac{1}{n}\sum_j \left[\log \det \left|\nabla f_{\mathbf{w}}\left(z_{\mathbf{w}}^j\right)\right| - \log P\left(z_{\mathbf{w}}^j\right)\right] \\ &= \hat{\mathbb{E}}_{x\sim P_{\rm data}}\left\{\log \det \left|\nabla f_{\mathbf{w}}\left(z_{\mathbf{w}}\right)\right| - \log P\left(z_{\mathbf{w}}\right)\right\}\end{aligned}$$

where $z_{\mathbf{w}}^j = f_{\mathbf{w}}^{-1}\left(x^j\right)$

+ *Is it an easy to differentiate risk then?*
- *Let's take a look!*

Flow-based Models: *MLE Learning*

Looking at the MLE risk function

$$\hat{R}(\mathbf{w}) = \hat{\mathbb{E}}_{x \sim P_{\text{data}}} \{\log \det |\nabla f_{\mathbf{w}}(z_{\mathbf{w}})| - \log P(z_{\mathbf{w}})\}$$

- *We do know* $P(z)$ ⇝ *we can write chain rule*

Backpropagation on $f_{\mathbf{w}}^{-1}$

$$\nabla_{\mathbf{w}} P(z_{\mathbf{w}}) = \nabla_z P(z_{\mathbf{w}}) \circ \boxed{\nabla_{\mathbf{w}} z_{\mathbf{w}}}$$

↳ *It's easily computed by standard backpropagation*

- *For first term, we use* $\nabla_{\mathbf{X}} \log \det |\mathbf{X}| = (\mathbf{X}^{-1})^{\mathsf{T}}$ *with chain rule to write*

$$\nabla_{\mathbf{w}} \log \det |\nabla f_{\mathbf{w}}(z_{\mathbf{w}})| = \left(\nabla f_{\mathbf{w}}(z_{\mathbf{w}})^{-1}\right)^{\mathsf{T}} \circ \nabla_{\mathbf{w}} [\nabla f_{\mathbf{w}}(z_{\mathbf{w}})]$$

↳ *It needs inverse of Jacobian and backpropagation over Jacobian*

↳ *It can be computationally expensive, though yet feasible*

Flow-based Models: *Training*

`Train_Flow(`$\mathbb{D}$`:dataset):`

Initiate the flow model $f_{\mathbf{w}}$ *with some* $\mathbf{w}$
for *multiple epochs* **do**
    Sample a batch of data samples $\left\{x^j : j = 1, \dots, n\right\}$ *from* $\mathbb{D}$
    for $j = 1, \dots, n$ **do**
        Invert flow to get latent representation $z^j \leftarrow f_{\mathbf{w}}^{-1}\left(x^j\right)$
        Compute $\hat{R}^j = \log \det \left|\nabla f_{\mathbf{w}}\left(z^j\right)\right| - \log P\left(z^j\right)$ *by forward passing* z^j
        Backpropagate to compute $\nabla_{\mathbf{w}} \hat{R}^j$
    end for
    Update $\mathbf{w}$ *using* `Opt_avg` $\left\{\nabla_{\mathbf{w}} \hat{R}^j\right\}$
end for
return *trained flow model* $f_{\mathbf{w}}$

Real-valued Non-Volume Preserving

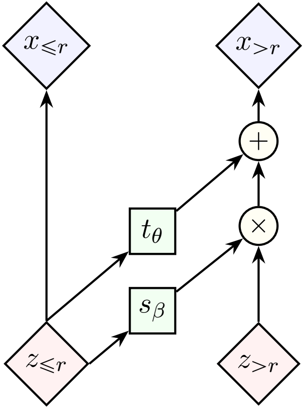
Real NVP develops *deep flow network for visual generation*

- *It uses a unit flow model for multiple steps*
  - ↳ *The inverse flow is of a dual form*
- *It starts with Gaussian latent*
- *Flow model is designed to keep training computationally easy*
  - ↳ *It's designed to keep the derivation of*

$$\hat{\mathbb{E}}_{x \sim P_{\text{data}}} \{\log \det |\nabla f_{\mathbf{w}}(z)|\}$$

simple

Real NVP: *Flow*



For some $r < d$*, let*

$$s_\beta : \mathbb{R}^r \mapsto \mathbb{R}^{d-r} \leftarrow \text{scaling}$$
$$t_\theta : \mathbb{R}^r \mapsto \mathbb{R}^{d-r} \leftarrow \text{translation}$$

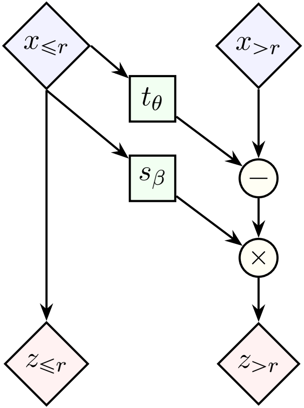
be two computational models: we build flow as

$$x_{\leqslant r} = z_{\leqslant r}$$
$$x_{>r} = z_{>r} \odot \exp\left\{ s_\beta \left(z_{\leqslant r} \right) \right\} + t_\theta \left(z_{\leqslant r} \right)$$

whose model parameters are $\mathbf{w} = [\theta, \beta]$*, i.e.,*

$$x = f_{\mathbf{w}}\left(z \right)$$

Real NVP: *Inverse Flow*



The flow is readily inverted

$$z_{\leqslant r} = x_{\leqslant r}$$
$$z_{>r} = \left(x_{>r} - t_\theta\left(x_{\leqslant r}\right)\right) \odot \exp\left\{-s_\beta\left(x_{\leqslant r}\right)\right\}$$

we can compactly show it as

$$z = f_{\mathbf{w}}^{-1}\left(x\right)$$

which is of dual form to direct flow $f_{\mathbf{w}}\left(z\right)$

Real NVP: *Jacobian*

By standard derivation: *we can show that*

$$\nabla f_{\mathbf{w}}\left(z\right) = \begin{bmatrix} \mathbf{I}_{d\times d} & \mathbf{0}_{d\times d-r} \\ \nabla_{x_{\leqslant r}} t_\theta\left(x_{\leqslant r}\right) & \begin{bmatrix} \exp\left\{s_1\right\} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \exp\left\{s_{d-r}\right\} \end{bmatrix} \end{bmatrix}$$

where $s_1, \dots, s_{d-r}$ are the outputs of scaling model, i.e.,

$$\texttt{DataType}\left\{s_1, \dots, s_{d-r}\right\} \leftarrow s_\beta\left(x_{\leqslant r}\right)$$

This is an upper-triangle matrix whose determinant $\equiv$ product of main diagonal

$$\det\left|\nabla f_{\mathbf{w}}\left(z\right)\right| = \prod_{j=1}^{d-r} \exp\left\{s_j\right\} = \exp\left\{\sum_j s_j\right\}$$

Real NVP: *Training*

As a result

$$\log \det |\nabla f_{\mathbf{w}}(z)| = \sum_j s_j = \texttt{sum}[s_\beta(x_{\leqslant r})]$$

whose gradient is readily given by

$$\nabla_{\mathbf{w}} \log \det |\nabla f_{\mathbf{w}}(z_{\mathbf{w}})| = \texttt{sum}[\nabla_\beta s_\beta(x_{\leqslant r})]$$

Also, if we consider standard Gaussian latent, we have

$$\log P(z) = -\frac{1}{2}\|z\|^2 - \textit{Constant}$$

and thus, the gradient is readily computed by backpropagation

$$\nabla_{\mathbf{w}} \log P(z) = -\frac{1}{2}\nabla \|f_{\mathbf{w}}^{-1}(x)\|^2$$

Real NVP: *Sample Output*

Data samples from CelebA

Samples from the flow network

Variants of Real NVP: *NICE*

Real NVP was indeed the extension of *an initial architecture*

NICE: Non-linear Independent Component Estimation (NICE)

which mainly had a simpler flow

$$x_{\leqslant r} = z_{\leqslant r}$$
$$x_{>r} = z_{>r} + t_\theta \left(z_{\leqslant r} \right)$$

This model was one of earliest computational flow-based models

- ✔ *It was simple to train due to simple risk function*
- ✘ *It was not too expressive, i.e., limited in capacity*

The latter issue was the reason for Real NVP proposal

Variants of Real NVP: *Glow*

Real NVP was later extended *to more flexible architecture called*

Glow: Generative Flow

in which permutation of data entries, i.e., x_i's in x, is learned as well

- *Glow uses 1×1 convolution to combine entries in the flow*
  - ↳ *This enables the possibility to learn how to couple channels*
  - ↳ *Remember that in Real NVP, the coupling is fixed*
    - ↳ *We fix choice of r and combine $x_{\leqslant r}$ with $x_{>r}$*
- *The training may need more computation*
- *Generation quality is generally better*

Some other known extensions are

- *FFJORD, Flow++, VFlow, Wavelet Flow*

Flow-based Models: *Wrap Up*

Flow-based models sound classic: *they are*

- *easy to sample* $\rightsquigarrow$ *pass noise sample* $z \sim P(z)$ *through flow network*
  - ↳ *Faster than AR models*
  - ↳ *Both faster and easier than EBMs*
- *imposing high computation for training, as we need to solve*

$$\min_{\mathbf{w}} \hat{\mathbb{E}}_{x \sim P_{\text{data}}} \left\{ \log \det \left| \nabla f_{\mathbf{w}}(z) \right| - \log P(z) \right\} \quad \textit{with} \quad z = f_{\mathbf{w}}^{-1}(x)$$

which can be computationally expensive

- ↳ *Generally, more complex than classic AR models*
- ↳ *Definitely easier than EBMs*

Final Notes

We are over with explicit methods: *in all these methods*

we directly targeted to learn data distribution

These three methods, i.e., AR, EBMs and Flow-based make the foundation

Make More Explicit Approaches!

We can combine these methods to come up with other approaches

- *In AR flow-based modeling, we learn autoregressive conditional distributions by flow-based models*

Though effective, these approaches limit us in various senses

! *In flow-based models we had to use same-dimension latent space*

↳ *We know that we can have much smaller latent space!*

We next study approaches in which we learn data distribution implicitly!