Deep generative modeling: ARMs and Normalizing Flows

Jakub M. Tomczak
Deep Learning

VRIJE UNIVERSITEIT AMSTERDAM

TYPES OF GENERATIVE MODELS

Generative model

- **Autoregressive** (*e.g.*, PixelCNN)
- **Flow-based** (e.g., RealNVP, GLOW)
- **Latent variable models**
  - **Implicit models** (*e.g.*, GANs)
  - **Prescribed models** (*e.g.*, VAE)

GENERATIVE MODELS

	Training	Likelihood	Sampling	Compression
Autoregressive models (e.g., PixelCNN)	Stable	Exact	Slow	No
Flow-based models (e.g., RealNVP)	Stable	Exact	Fast/Slow	No
Implicit models (e.g., GANs)	Unstable	No	Fast	No
Prescribed models (e.g., VAEs)	Stable	Approximate	Fast	Yes

GENERATIVE MODELS

	Training	Likelihood	Sampling	Compression
Autoregressive models (e.g., PixelCNN)	Stable	Exact	Slow	No
Flow-based models (e.g., RealNVP)	Stable	Exact	Fast/Slow	No
Implicit models (e.g., GANs)	Unstable	No	Fast	No
Prescribed models (e.g., VAEs)	Stable	Approximate	Fast	Yes

ARMS: AUTOREGRESSIVE MODELS

REPRESENTING A JOINT DISTRIBUTION

There are two main rules in the probability theory:

- Sum rule: $p(x) = \sum_{y} p(x, y)$
- Product rule: $p(x, y) = p(y \mid x)\ p(x)$

REPRESENTING A JOINT DISTRIBUTION

There are two main rules in the probability theory:

- Sum rule: $p(x) = \sum_{y} p(x, y)$
- Product rule: $p(x, y) = p(y \mid x)\ p(x)$

Before, we used these two rules for latent-variable models:

$$
\begin{aligned}
p(\mathbf{x}) &= \int p(\mathbf{x}, \mathbf{z})\ \mathrm{d}\mathbf{z} \\
&= \int p(\mathbf{x} \mid \mathbf{z})\ p(\mathbf{z})\ \mathrm{d}\mathbf{z}
\end{aligned}
$$

There are two main rules in the probability theory:

- Sum rule: $p(x) = \sum_{y} p(x, y)$
- Product rule: $p(x, y) = p(y | x)\ p(x)$

Now, we will use the product rule to express the distribution of $\mathbf{x} \in \mathbb{R}^D$:

$$p(\mathbf{x}) = p(x_1) \sum_{d=2}^{D} p(x_d | \mathbf{x}_{<d})$$

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]^\top$

REPRESENTING A JOINT DISTRIBUTION

We use the product rule to express the distribution of $\mathbf{x} \in \mathbb{R}^D$:

$$p(\mathbf{x}) = p(x_1) \sum_{d=2}^{D} p(x_d | \mathbf{x}_{<d})$$

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]^\top$.

REPRESENTING A JOINT DISTRIBUTION

We use the product rule to express the distribution of $\mathbf{x} \in \mathbb{R}^D$:

$$p(\mathbf{x}) = p(x_1) \sum_{d=2}^{D} p(x_d | \mathbf{x}_{<d})$$

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]^\top$.

The order of variables isn't important.

REPRESENTING A JOINT DISTRIBUTION

We use the product rule to express the distribution of $\mathbf{x} \in \mathbb{R}^D$:

$$p(\mathbf{x}) = p(x_1) \sum_{d=2}^{D} p(x_d | \mathbf{x}_{<d})$$

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]^\top$.

The order of variables isn't important.

However, modeling all conditionals separately is **infeasible**…

REPRESENTING A JOINT DISTRIBUTION

We use the product rule to express the distribution of $\mathbf{x} \in \mathbb{R}^D$:

$$p(\mathbf{x}) = p(x_1) \sum_{d=2}^{D} p(x_d | \mathbf{x}_{<d})$$

where $\mathbf{x}_{<d} = [x_1, x_2, \ldots, x_{d-1}]^\top$.

The order of variables isn't important.

However, modeling all conditionals separately is **infeasible**…

Can we do better that?

We can assume a **finite dependency**.

We can assume a **finite dependency**.

For instance, for two last variables:

$$p(\mathbf{x}) = p(x_1)p(x_2 | x_1) \sum_{d=3}^{D} p(x_d | x_{d-2}, x_{d-1})$$

Now, we can model $p(x_d | x_{d-2}, x_{d-1})$ by a **single model**.

➡E.g., we can take a **neural network**.

We can assume a **finite dependency**.

For instance, for two last variables:

$$p(\mathbf{x}) = p(x_1)p(x_2 | x_1) \sum_{d=3}^{D} p(x_d | x_{d-2}, x_{d-1})$$

REPRESENTING A JOINT DISTRIBUTION

We can assume a **finite dependency**.

For instance, for two last variables:

$$p(\mathbf{x}) = p(x_1)p(x_2 | x_1) \sum_{d=3}^{D} p(x_d | x_{d-2}, x_{d-1})$$

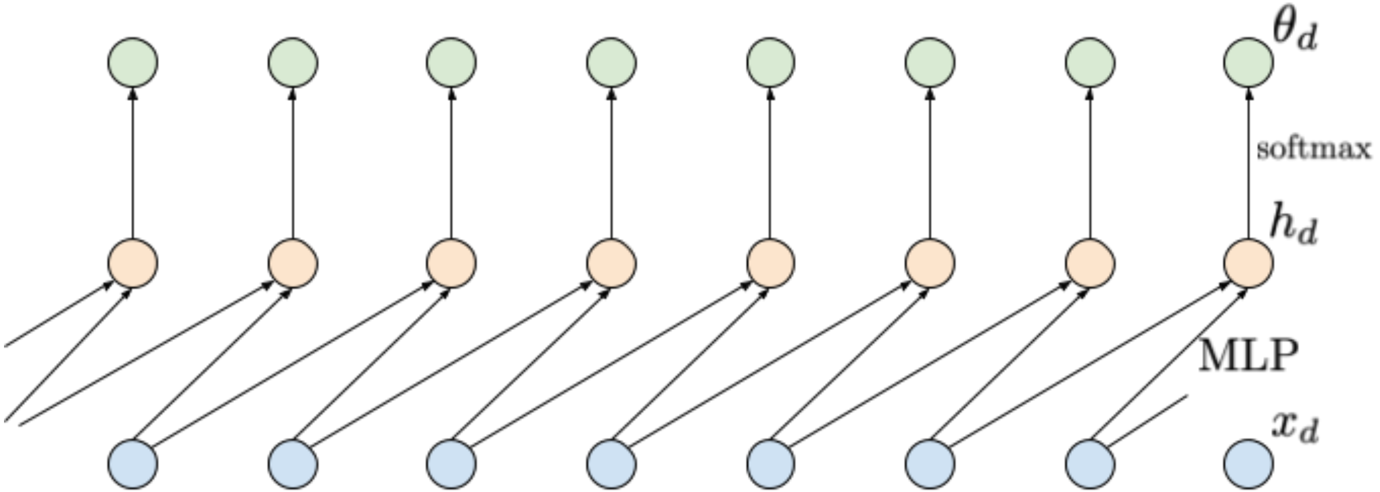
Now, we can model $p(x_d | x_{d-2}, x_{d-1})$ by a **single model**.

➡E.g., we can take a **neural network**.

We can assume a **finite dependency**.

For instance, for two last variables:

$$p(\mathbf{x}) = p(x_1)p(x_2 | x_1) \sum_{d=3}^{D} p(x_d | x_{d-2}, x_{d-1})$$



We can assume a **finite dependency**.

For instance, for two last variables:

$$p(\mathbf{x}) = p(x_1)p(x_2 | x_1) \sum_{d=3}^{D} p(x_d | x_{d-2}, x_{d-1})$$

Now, we can model $p(x_d | x_{d-2}, x_{d-1})$ by a **single model**.

➡E.g., we can take a **neural network**.

However, it is still pretty **limiting**, because we need to decide on the length of the dependency.

AUTOREGRESSIVE MODELS (ARM)

Instead, we can use RNNs to model the conditionals:

$$p(x_d | \mathbf{x}_{<d}) = p\left(x_d | RNN(x_{d-1}, h_{d-1})\right)$$

where $h_d = RNN(x_{d-1}, h_{d-1})$.

AUTOREGRESSIVE MODELS (ARM)

Instead, we can use RNNs to model the conditionals:

$$p(x_d | \mathbf{x}_{<d}) = p\left(x_d | RNN(x_{d-1}, h_{d-1})\right)$$

where $h_d = RNN(x_{d-1}, h_{d-1})$.

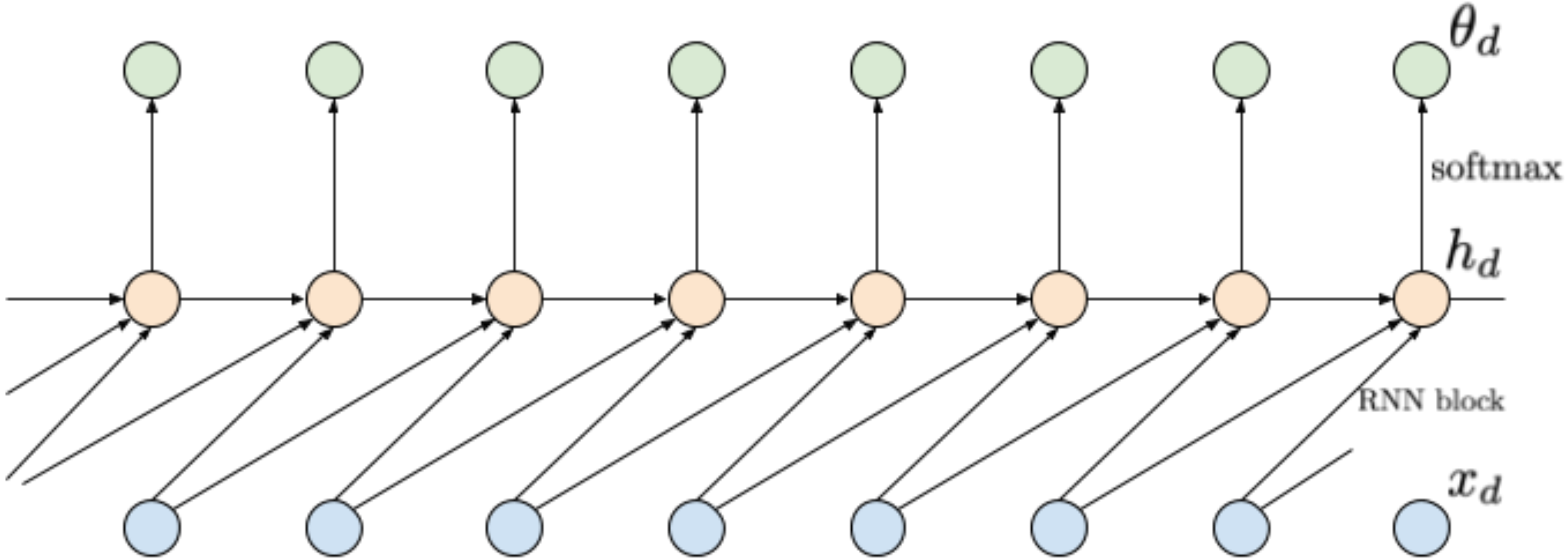
Advantages:

➡ We don’t need to define dependencies.

➡ A **single parameterization**.

Instead, we can use RNNs to model the conditionals:

$$p(x_d|\mathbf{x}_{<d}) = p\left(x_d|RNN(x_{d-1}, h_{d-1})\right)$$



Instead, we can use RNNs to model the conditionals:

$$p(x_d | \mathbf{x}_{<d}) = p\left(x_d | RNN(x_{d-1}, h_{d-1})\right)$$

where $h_d = RNN(x_{d-1}, h_{d-1})$.

Advantages:

➡ We don’t need to define dependencies.

➡ A **single parameterization**.

RNN are slow, because they’re sequential.

Instead, we can use RNNs to model the conditionals:

$$p(x_d | \mathbf{x}_{<d}) = p\left(x_d | RNN(x_{d-1}, h_{d-1})\right)$$

where $h_d = RNN(x_{d-1}, h_{d-1})$.

Advantages:

➡ We don’t need to define dependencies.

➡ A **single parameterization**.

RNN are slow, because they’re sequential. Can we do better?

Let us consider a sequence $\mathbf{x} = [x_1, x_2, \ldots, x_D]^\top$.

We assume all observed data are *D*-dimensional.

USING CONVOLUTIONAL NEURAL NETWORKS FOR MODELING SEQUENCES

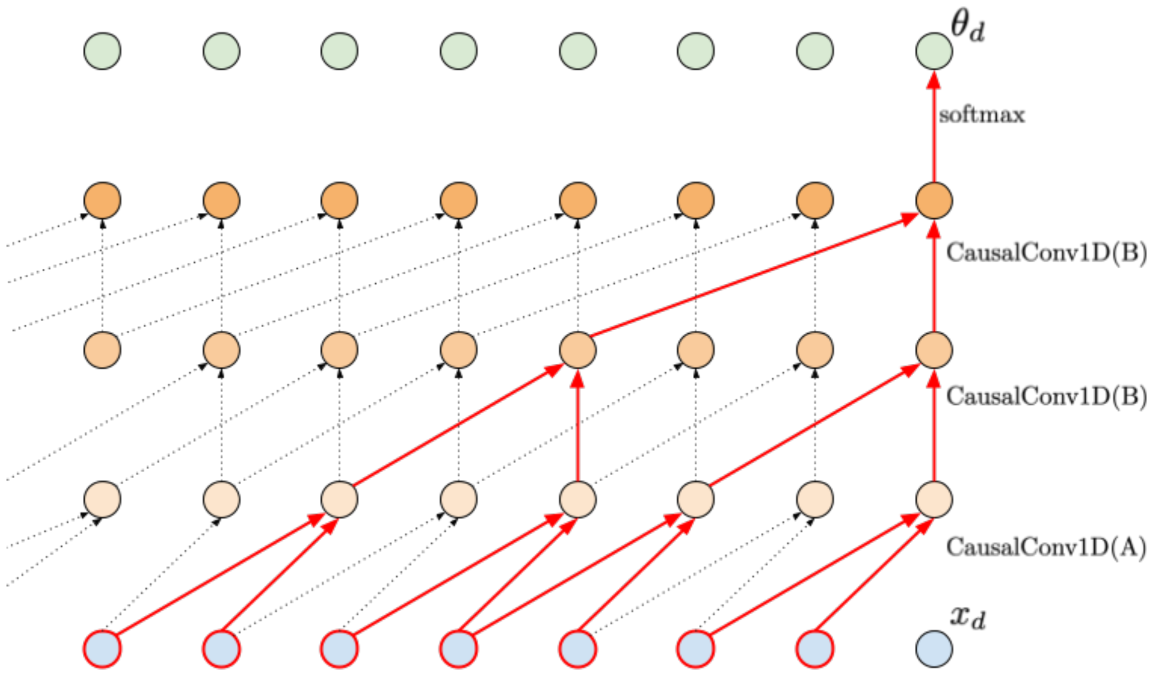
Let us consider a sequence $\mathbf{x} = [x_1, x_2, \ldots, x_D]^\top$.

We assume all observed data are *D*-dimensional.

We can use **1*D* convolutional layers** to process all signals at once.

Moreover, we can use **dilation** to learn **long-range dependencies**.

Let us consider a sequence $\mathbf{x} = [x_1, x_2, \ldots, x_D]^\top$.

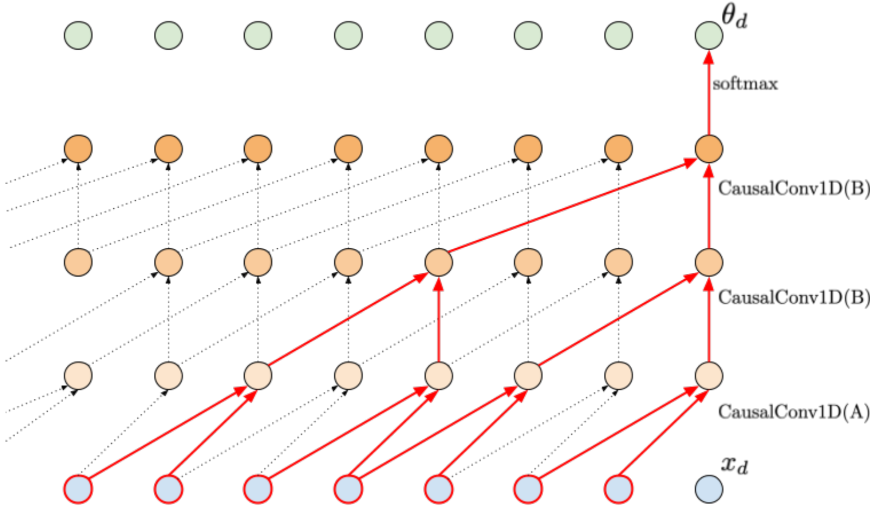


We can use **1*D* convolutional layers** to process all signals at once.

Notice:

- **Causal convolution**

  (i.e., looking only to the "past")

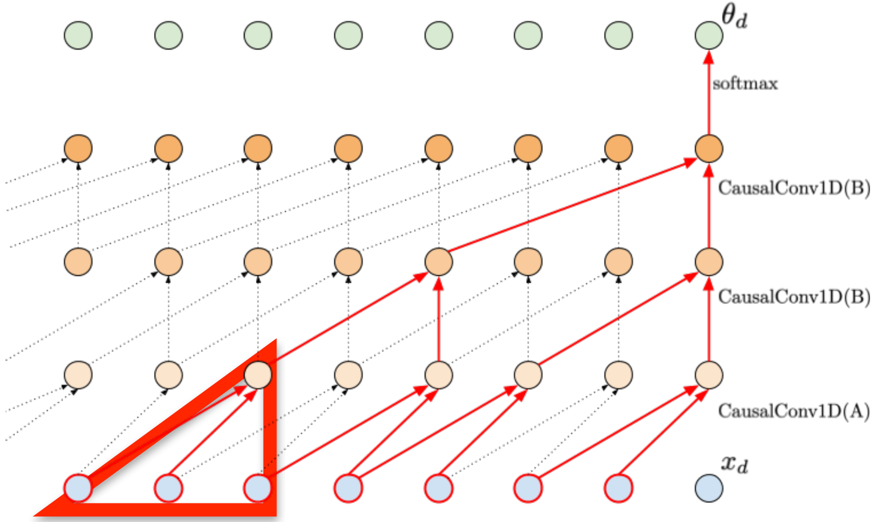


We can use **1*D* convolutional layers** to process all signals at once.

Notice:

- **Causal convolution**

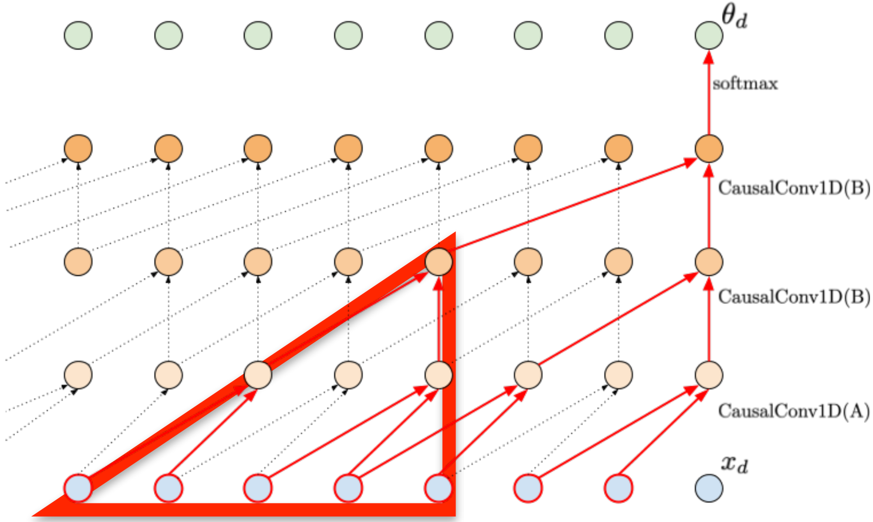
  (i.e., looking only to the “past”)



We can use **1*D* convolutional layers** to process all signals at once.

Notice:

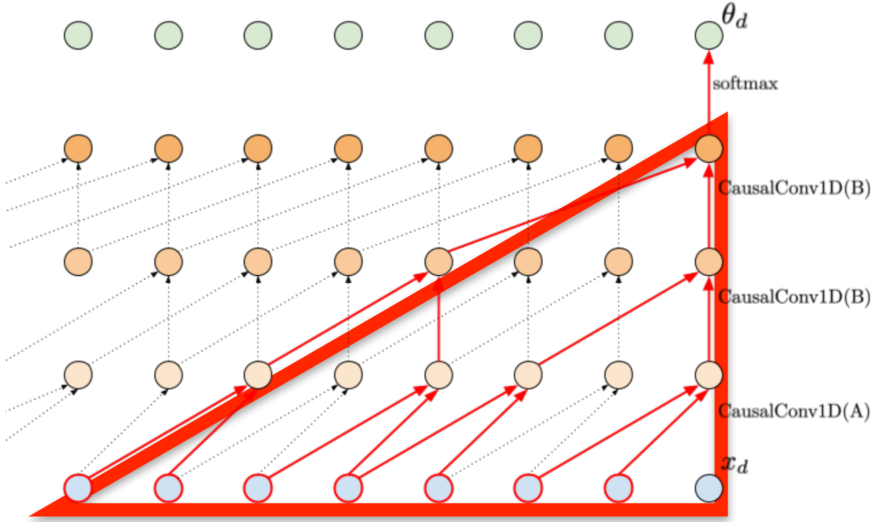
- **Causal convolution**
  (i.e., looking only to the "past")



We can use **1*D* convolutional layers** to process all signals at once.

Notice:

- **Causal convolution**
  (i.e., looking only to the "past")



USING CONVOLUTIONAL NEURAL NETWORKS FOR MODELING SEQUENCES

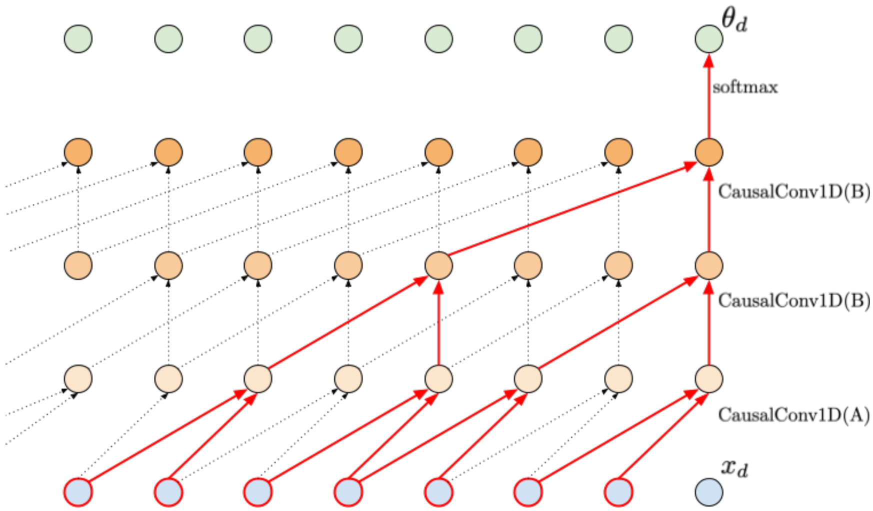
We can use **1*D* convolutional layers** to process all signals at once.

Notice:

- **Causal convolution**

  (i.e., looking only to the "past")

- Very **efficient** using current DL frameworks.

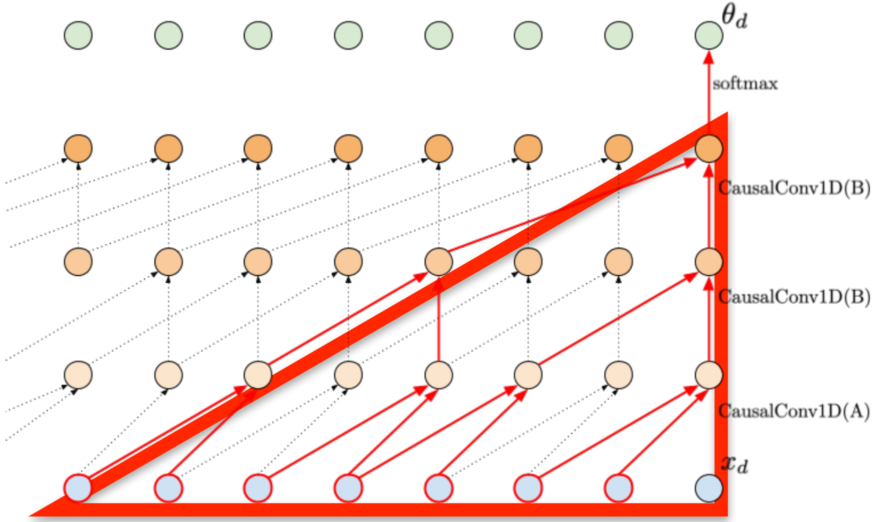


USING CONVOLUTIONAL NEURAL NETWORKS FOR MODELING SEQUENCES

We can use **1*D* convolutional layers** to process all signals at once.

Notice:

- **Causal convolution**
  (i.e., looking only to the "past")
- Very **efficient** using current DL frameworks.



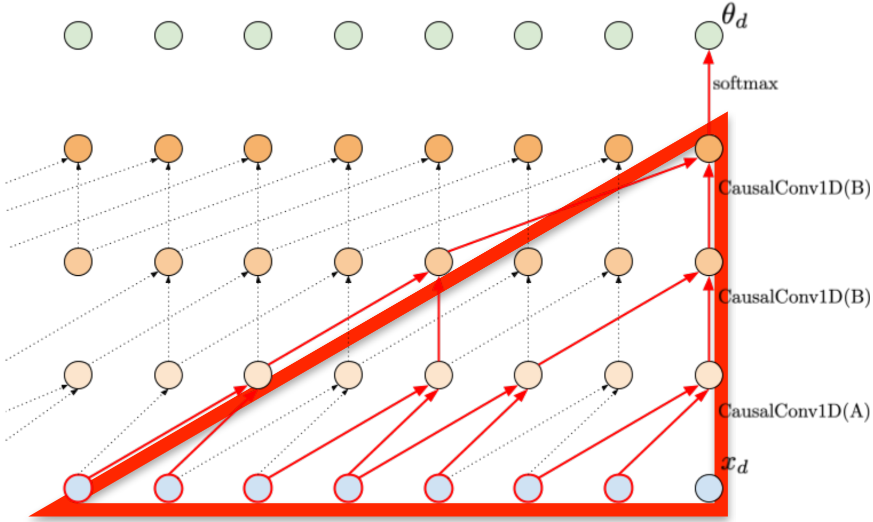
- For deep neural networks, NNs learn long-range dependencies.

USING CONVOLUTIONAL NEURAL NETWORKS FOR MODELING SEQUENCES

We can use **1*D* convolutional layers** to process all signals at once.

Notice:

- **Causal convolution**
  (i.e., looking only to the "past")
- Very **efficient** using current DL frameworks.



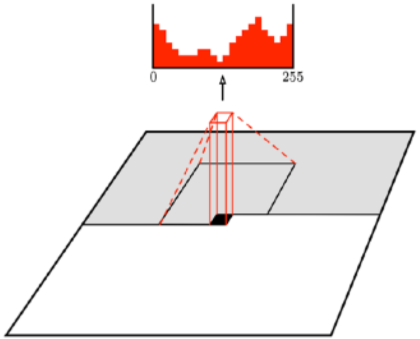
- For deep neural networks, NNs learn long-range dependencies.

 Oord, Aaron van den, et al. "**Wavenet**: A generative model for raw audio." *arXiv preprint arXiv:1609.03499* (2016).

PIXELCNN

We can utilize the very same idea for images, but using **2D convolutions**.

We need to remember about **causal convolutions**!

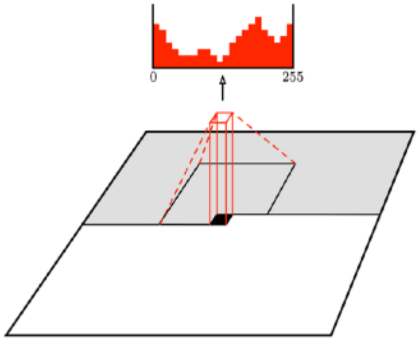


1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

We can utilize the very same idea for images, but using **2D convolutions**.

We need to remember about **causal convolutions**!

Moreover, we should think of 2 or even **3 dimensions** (CxHxW).

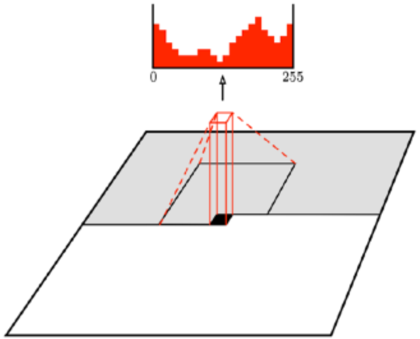


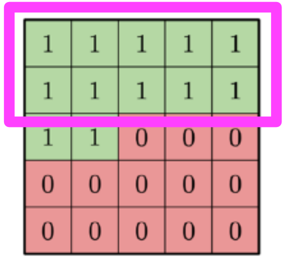
1	1	1	1	1
1	1	1	1	1
1	1	0	0	0
0	0	0	0	0
0	0	0	0	0

We can accomplish that by composing **two Conv2D layers**.

We can utilize the very same idea for images, but using **2D convolutions**.

We need to remember about **causal convolutions**!





Moreover, we should think of 2 or even **3 dimensions** (CxHxW).

We can accomplish that by composing **two Conv2D layers**.

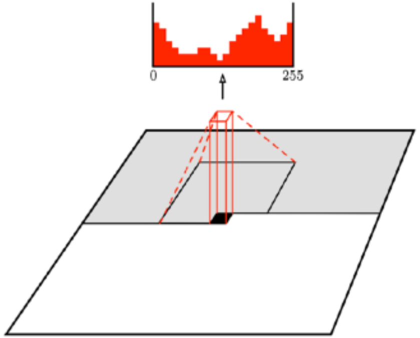
The first Conv2D layer covers the upper part.

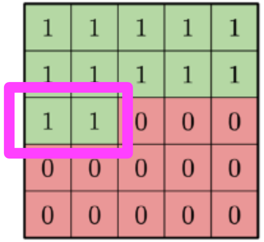
PIXELCNN

We can utilize the very same idea for images, but using **2D convolutions**.

We need to remember about **causal convolutions**!

Moreover, we should think of 2 or even **3 dimensions** (CxHxW).





We can accomplish that by composing **two Conv2D layers**.

The first Conv2D layer covers the upper part.

The second Conv2D layer covers the left part.

 Van den Oord, Aaron, et al. "Conditional image generation with **PixelCNN** decoders." *NIPS*. 2016.

PIXELCNN

We can utilize the very same idea for images, but using **2D convolutions**.

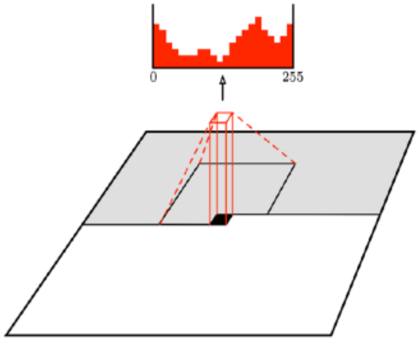
We need to remember about **causal convolutions**!

Moreover, we should think of 2 or even **3 dimensions** (CxHxW).

We can accomplish that by composing **two Conv2D layers**.

The first Conv2D layer covers the upper part.

The second Conv2D layer covers the left part.



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

using masking for kernel weights.

Van den Oord, Aaron, et al. "Conditional image generation with **PixelCNN** decoders." *NIPS*. 2016.

PIXELCNN

Originally, PixelCNN used the **softmax non-linearity** at the end to output integers between 0 and 255 (i.e., pixel values).

Currently, a **mixture of discretized logistic distributions** is used:

$$P(x \mid \pi, \mu, s) = \sum_{i=1}^{K} \pi_i \left[\sigma\left(\left(x + 0.5 - \mu_i \right) / s_i \right) - \sigma\left(\left(x - 0.5 - \mu_i \right) / s_i \right) \right]$$

Salimans, Tim, et al. "PixelCNN++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications." *arXiv* (2017).

PIXELCNN

Originally, PixelCNN used the **softmax non-linearity** at the end to output integers between 0 and 255 (i.e., pixel values).

Currently, a **mixture of discretized logistic distributions** is used:

$$P(x \mid \pi, \mu, s) = \sum_{i=1}^{K} \pi_i \left[\sigma\left((x + 0.5 - \mu_i)/s_i \right) - \sigma\left((x - 0.5 - \mu_i)/s_i \right) \right]$$

sigmoid function

Salimans, Tim, et al. "PixelCNN++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications." *arXiv* (2017).

Originally, PixelCNN used the **softmax non-linearity** at the end to output integers between 0 and 255 (i.e., pixel values).

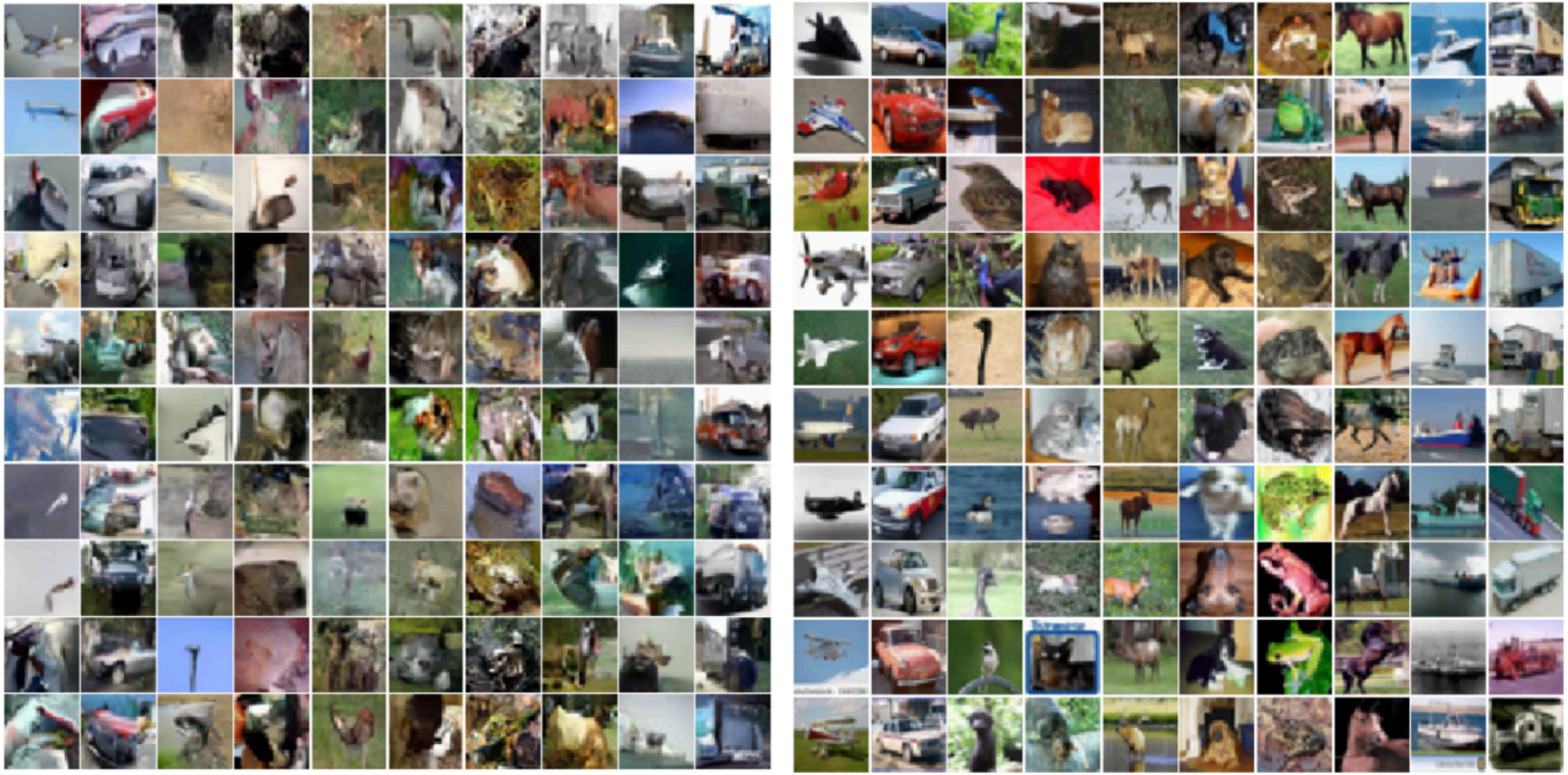
Currently, a **mixture of discretized logistic distributions** is used:

$$P(x \mid \pi, \mu, s) = \sum_{i=1}^{K} \pi_i \left[\sigma\Big(\big(x + 0.5 - \mu_i\big) / s_i \Big) - \sigma\Big(\big(x - 0.5 - \mu_i\big) / s_i \Big) \right]$$

Learnable as a parameter

 Salimans, Tim, et al. "PixelCNN++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications." *arXiv* (2017).

PIXELCNN



Sampled Real

42 Salimans, Tim, et al. "PixelCNN++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications." *arXiv* (2017).

Advantages

✓ Exact likelihood.

✓ Stable training.

Disadvantages

- Very slow sampling.
- No compression.
- Sometimes, low visual quality.

FLOW-BASED MODELS

Let us consider a random variable $v \in \mathbb{R}$ with $p(v) = \mathcal{N}(v \mid 0,1)$.

Let us consider a random variable $v \in \mathbb{R}$ with $p(v) = \mathcal{N}(v \mid 0,1)$.

Then, we take the following transformation: $u = 0.75 \cdot v + 1$.

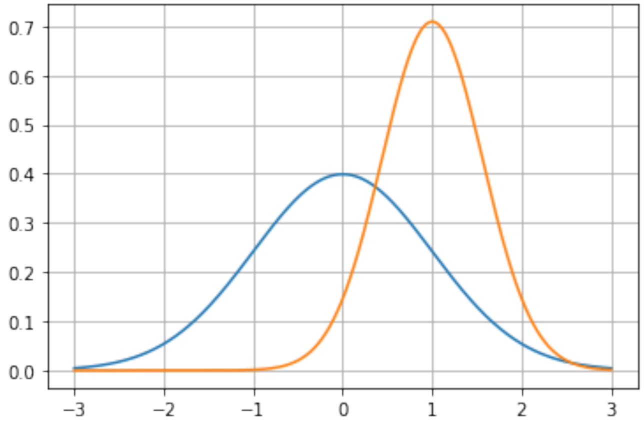
Q: What is the pdf for u?

Let us consider a random variable $v \in \mathbb{R}$ with $p(v) = \mathcal{N}(v \mid 0,1)$.

Then, we take the following transformation: $u = 0.75 \cdot v + 1$.

Q: What is the pdf for u?

A: The pdf is $p(u) = \mathcal{N}(u \mid 1, 0.75^2)$.

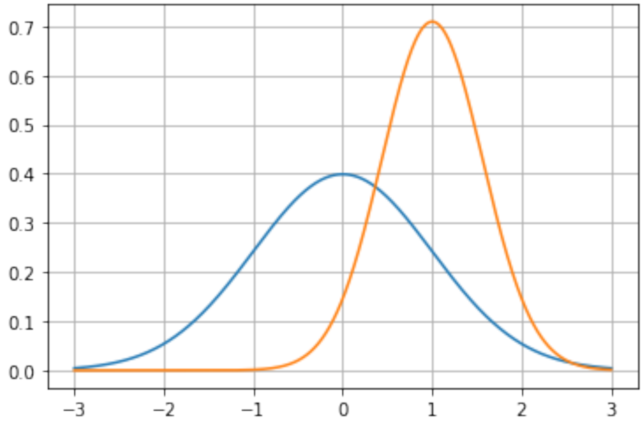


Let us consider a random variable $v \in \mathbb{R}$ with $p(v) = \mathcal{N}(v\,|\,0,1)$.

Then, we take the following transformation: $u = 0.75 \cdot v + 1$.

Q: What is the pdf for u?

A: The pdf is $p(u) = \mathcal{N}(u\,|\,1,0.75^2)$.



In general, we have:

$$p(u) = p\left(f^{-1}(u)\right)\left|\frac{\partial f^{-1}(u)}{\partial u}\right|$$

Let us consider a random variable $v \in \mathbb{R}$ with $p(v) = \mathcal{N}(v \mid 0,1)$.

Then, we take the following transformation: $u = 0.75 \cdot v + 1$.

Q: What is the pdf for u?

A: The pdf is $p(u) = \mathcal{N}(u \mid 1, 0.75^2)$.

In general, we have:

$$p(u) = p\left(f^{-1}(u)\right) \left| \frac{\partial f^{-1}(u)}{\partial u} \right|$$

$$f^{-1}(u) = \frac{u-1}{0.75}$$

$$\left| \frac{\partial f^{-1}(u)}{\partial u} \right| = \frac{4}{3}$$

Let us consider a random variable $v \in \mathbb{R}$ with $p(v) = \mathcal{N}(v \,|\, 0,1)$.

Then, we take the following transformation: $u = 0.75 \cdot v + 1$.

Q: **What is the pdf for u?**

$$f^{-1}(u) = \frac{u-1}{0.75}$$

A: The pdf is $p(u) = \mathcal{N}(u \,|\, 1, 0.75^2)$.

$$\left| \frac{\partial f^{-1}(u)}{\partial u} \right| = \frac{4}{3}$$

In general, we have:

$$p(u) = p\left(f^{-1}(u)\right) \left| \frac{\partial f^{-1}(u)}{\partial u} \right|$$

$$p(u) = p\left(\frac{u-1}{0.75}\right)\frac{4}{3} = \frac{1}{\sqrt{2\pi\ 0.75^2}} \exp\left\{-(u-1)^2/0.75^2\right\}$$

CHANGE OF VARIABLES

Multidimensional case:

$$p(\mathbf{u}) = p\left(f^{-1}(\mathbf{u})\right)\left|\frac{\partial f^{-1}(\mathbf{u})}{\partial \mathbf{u}}\right|$$

where:

$$\left|\frac{\partial f^{-1}(\mathbf{u})}{\partial \mathbf{u}}\right| = \left|\det \mathbf{J}_{f^{-1}(\mathbf{u})}\right|$$

Multidimensional case:

$$p(\mathbf{u}) = p\left(f^{-1}(\mathbf{u})\right) \left| \frac{\partial f^{-1}(\mathbf{u})}{\partial \mathbf{u}} \right|$$

where:

$$\left| \frac{\partial f^{-1}(\mathbf{u})}{\partial \mathbf{u}} \right| = \left| \det \mathbf{J}_{f^{-1}(\mathbf{u})} \right|$$

Jacobian

$$\mathbf{J}_{f^{-1}} = \begin{bmatrix} \frac{\partial f_1^{-1}}{\partial u_1} & \cdots & \frac{\partial f_1^{-1}}{\partial u_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D^{-1}}{\partial u_1} & \cdots & \frac{\partial f_D^{-1}}{\partial u_D} \end{bmatrix}$$

Multidimensional case:

$$p(\mathbf{u}) = p\left(f^{-1}(\mathbf{u})\right) \left| \frac{\partial f^{-1}(\mathbf{u})}{\partial \mathbf{u}} \right|$$

where:

$$\left| \frac{\partial f^{-1}(\mathbf{u})}{\partial \mathbf{u}} \right| = \left| \det \mathbf{J}_{f^{-1}(\mathbf{u})} \right|$$

Jacobian

$$\mathbf{J}_{f^{-1}} = \begin{bmatrix} \frac{\partial f_1^{-1}}{\partial u_1} & \cdots & \frac{\partial f_1^{-1}}{\partial u_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_D^{-1}}{\partial u_1} & \cdots & \frac{\partial f_D^{-1}}{\partial u_D} \end{bmatrix}$$

How can we utilize this idea?

APPLYING CHANGE OF VARIABLES AND INVERTIBLE TRANSFORMATIONS

Let us consider a sequence of invertible transformations $f_k : \mathbb{R}^D \to \mathbb{R}^D$.

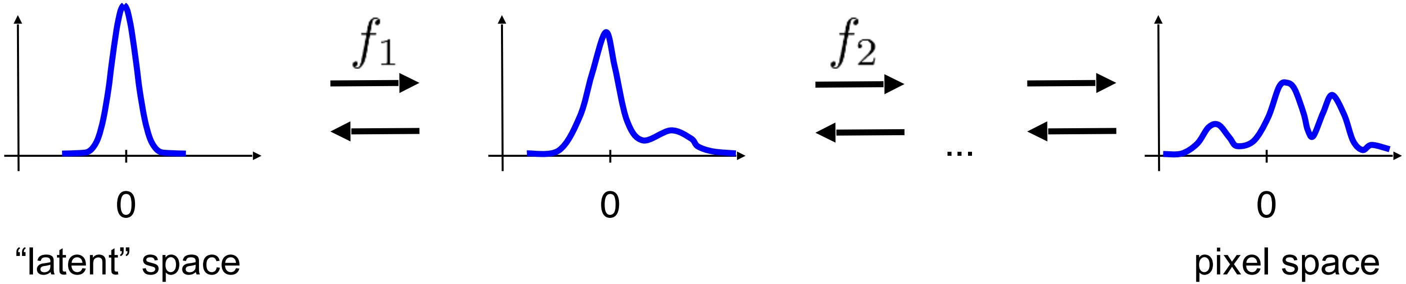
We can start with a *simple* distribution, e.g., $\pi(\mathbf{z}) = \mathcal{N}(\mathbf{z} \,|\, 0, \mathbf{I})$.

Rippel, O., & Adams, R. P. (2013). High-dimensional probability estimation with deep density models. arXiv

APPLYING CHANGE OF VARIABLES AND INVERTIBLE TRANSFORMATIONS

Let us consider a sequence of invertible transformations $f_k : \mathbb{R}^D \rightarrow \mathbb{R}^D$.

We can start with a *simple* distribution, e.g., $\pi(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$.

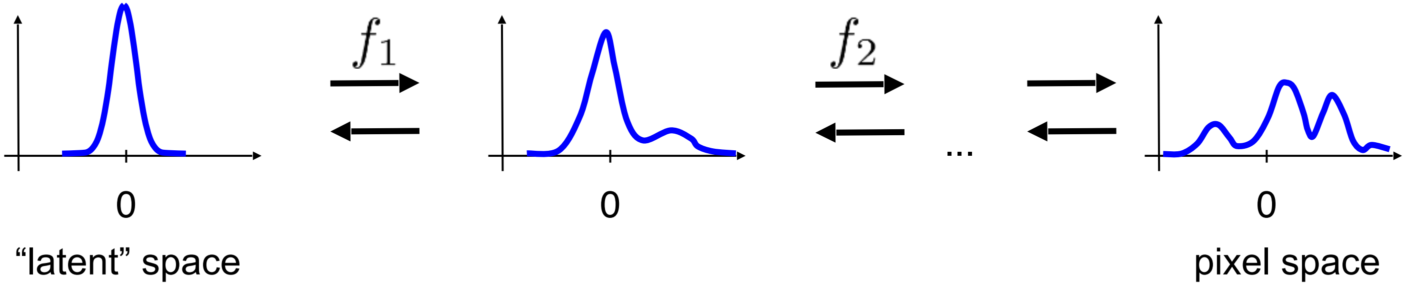


Rippel, O., & Adams, R. P. (2013). High-dimensional probability estimation with deep density models. arXiv

APPLYING CHANGE OF VARIABLES AND INVERTIBLE TRANSFORMATIONS

Let us consider a sequence of invertible transformations $f_k : \mathbb{R}^D \to \mathbb{R}^D$.

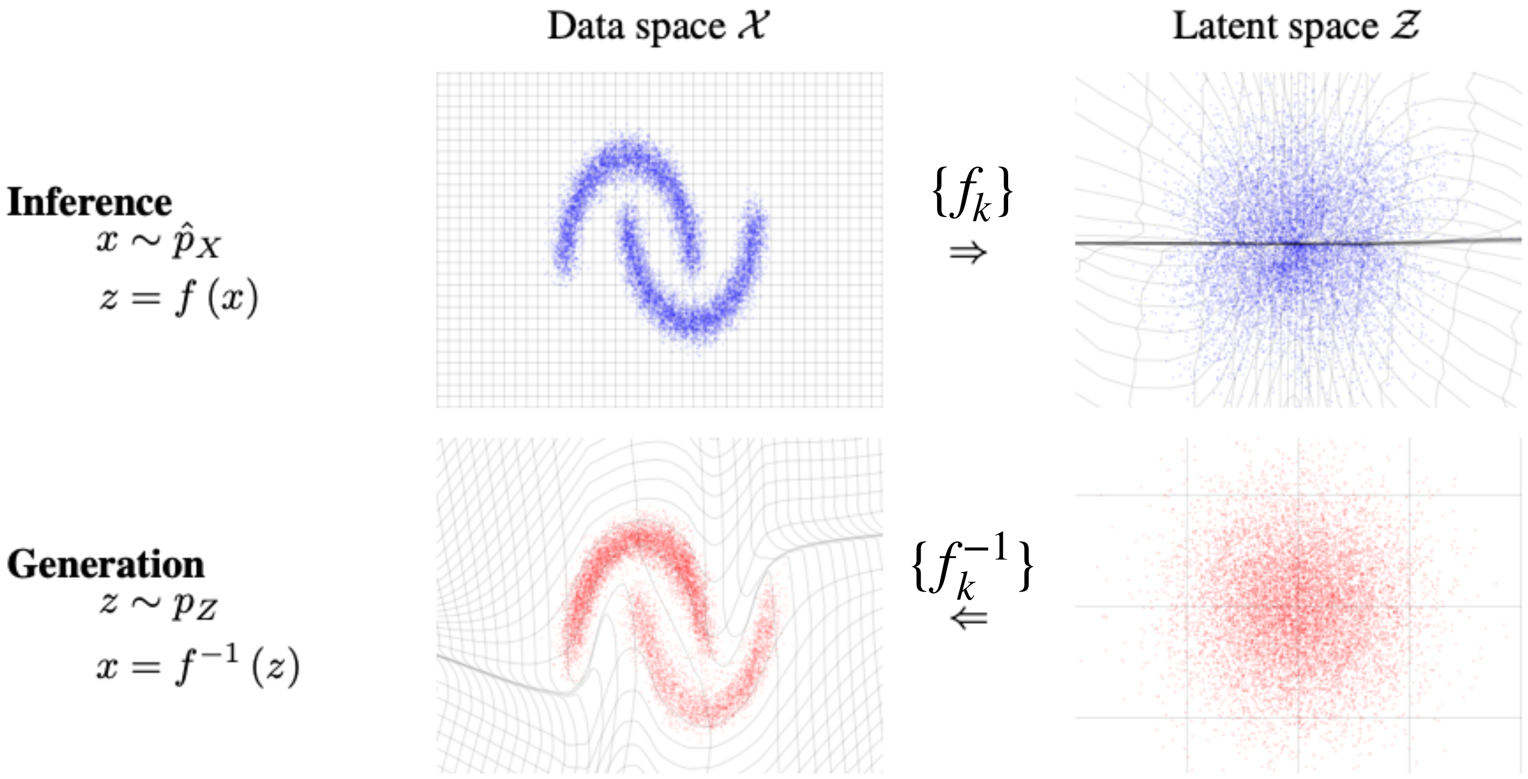
We can start with a *simple* distribution, e.g., $\pi(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, \mathbf{I})$.



This results in: $p(\mathbf{x}) = \pi\left(\mathbf{z}_0\right) \prod_{i=1}^{K} \left| \det \frac{\partial f_i\left(\mathbf{z}_{i-1}\right)}{\partial \mathbf{z}_{i-1}} \right|^{-1}$

Rippel, O., & Adams, R. P. (2013). High-dimensional probability estimation with deep density models. arXiv

2D EXAMPLE


Data space $\mathcal{X}$
Latent space $\mathcal{Z}$
Inference
$x \sim \hat{p}_X$
$z = f(x)$
$\{f_k\}$
$\Rightarrow$
Generation
$z \sim p_Z$
$x = f^{-1}(z)$
$\{f_k^{-1}\}$
$\Leftarrow$

The density model: $p(\mathbf{x}) = \pi\left(\mathbf{z}_0\right) \prod_{i=1}^{K} \left| \det \frac{\partial f_i\left(\mathbf{z}_{i-1}\right)}{\partial \mathbf{z}_{i-1}} \right|^{-1}$

The density model: $p(\mathbf{x}) = \pi\left(\mathbf{z}_0\right) \prod_{i=1}^{K} \left| \det \frac{\partial f_i\left(\mathbf{z}_{i-1}\right)}{\partial \mathbf{z}_{i-1}} \right|^{-1}$

In order to obtain flexible transformations f_k, we use **neural networks**.

However, we need to ensure that they are **invertible**.

The density model: $p(\mathbf{x}) = \pi\left(\mathbf{z}_0\right) \prod_{i=1}^{K} \left| \det \frac{\partial f_i\left(\mathbf{z}_{i-1}\right)}{\partial \mathbf{z}_{i-1}} \right|^{-1}$

In order to obtain flexible transformations f_k, we use **neural networks**.

However, we need to ensure that they are **invertible**.

Moreover, we cannot use any invertible neural network, because we need to remember about the **Jacobian**!

The density model: $p(\mathbf{x}) = \pi\left(\mathbf{z}_0\right) \prod_{i=1}^{K} \left| \det \frac{\partial f_i\left(\mathbf{z}_{i-1}\right)}{\partial \mathbf{z}_{i-1}} \right|^{-1}$

In order to obtain flexible transformations f_k, we use **neural networks**.

However, we need to ensure that they are **invertible**.

Moreover, we cannot use any invertible neural network, because we need to remember about the **Jacobian**!

Calculating Jacobian is the main challenge in flow-based models.

REALNVP

Design the invertible transformations as follows:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$

$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp\Big(s\left(\mathbf{x}_{1:d}\right)\Big) + t\left(\mathbf{x}_{1:d}\right)$$

where: $s(\cdot)$ and $t(\cdot)$ are **arbitrary** neural networks.

Dinh, L., Sohl-Dickstein, J., & Bengio, S. (2016). Density estimation using **RealNVP**. arXiv preprint arXiv:1605.08803.

Design the invertible transformations as follows:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$

$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp\left(s\left(\mathbf{x}_{1:d}\right)\right) + t\left(\mathbf{x}_{1:d}\right)$$

where: $s(\cdot)$ and $t(\cdot)$ are **arbitrary** neural networks.

This is **invertible by design**, because:

$$\mathbf{x}_{d+1:D} = \left(\mathbf{y}_{d+1:D} - t\left(\mathbf{y}_{1:d}\right)\right) \odot \exp\left(-s\left(\mathbf{y}_{1:d}\right)\right)$$

$$\mathbf{x}_{1:d} = \mathbf{y}_{1:d}$$

Dinh, L., Sohl-Dickstein, J., & Bengio, S. (2016). Density estimation using **RealNVP**. arXiv preprint arXiv:1605.08803.

Design the invertible transformations as follows:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$

$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp\left(s\left(\mathbf{x}_{1:d}\right)\right) + t\left(\mathbf{x}_{1:d}\right)$$

Known as
affine coupling layer

where: $s(\cdot)$ and $t(\cdot)$ are **arbitrary** neural networks.

This is **invertible by design**, because:

$$\mathbf{x}_{d+1:D} = \left(\mathbf{y}_{d+1:D} - t\left(\mathbf{y}_{1:d}\right)\right) \odot \exp\left(-s\left(\mathbf{y}_{1:d}\right)\right)$$

$$\mathbf{x}_{1:d} = \mathbf{y}_{1:d}$$

Dinh, L., Sohl-Dickstein, J., & Bengio, S. (2016). Density estimation using **RealNVP**. arXiv preprint arXiv:1605.08803.

The invertible transformation:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$

$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp\left(s\left(\mathbf{x}_{1:d}\right)\right) + t\left(\mathbf{x}_{1:d}\right)$$

What about the Jacobian?

The invertible transformation:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$

$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp\left(s\left(\mathbf{x}_{1:d}\right)\right) + t\left(\mathbf{x}_{1:d}\right)$$

What about the Jacobian?

$$\mathbf{J} = \begin{bmatrix} \mathbb{I}_d & \mathbf{0}_{d\times(D-d)} \\ \frac{\partial \mathbf{y}_{d+1:D}}{\partial \mathbf{x}_{1:d}} & \mathrm{diag}(\exp(s(\mathbf{x}_{1:d}))) \end{bmatrix}$$

The invertible transformation:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$

$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp\left(s\left(\mathbf{x}_{1:d}\right)\right) + t\left(\mathbf{x}_{1:d}\right)$$

What about the Jacobian?

$$\mathbf{J} = \begin{bmatrix} \mathbb{I}_d & \mathbf{0}_{d\times(D-d)} \\ \frac{\partial \mathbf{y}_{d+1:D}}{\partial \mathbf{x}_{1:d}} & \mathrm{diag}(\exp(s(\mathbf{x}_{1:d}))) \end{bmatrix}$$

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

The invertible transformation:

$$\mathbf{y}_{1:d} = \mathbf{x}_{1:d}$$

$$\mathbf{y}_{d+1:D} = \mathbf{x}_{d+1:D} \odot \exp\left(s\left(\mathbf{x}_{1:d}\right)\right) + t\left(\mathbf{x}_{1:d}\right)$$

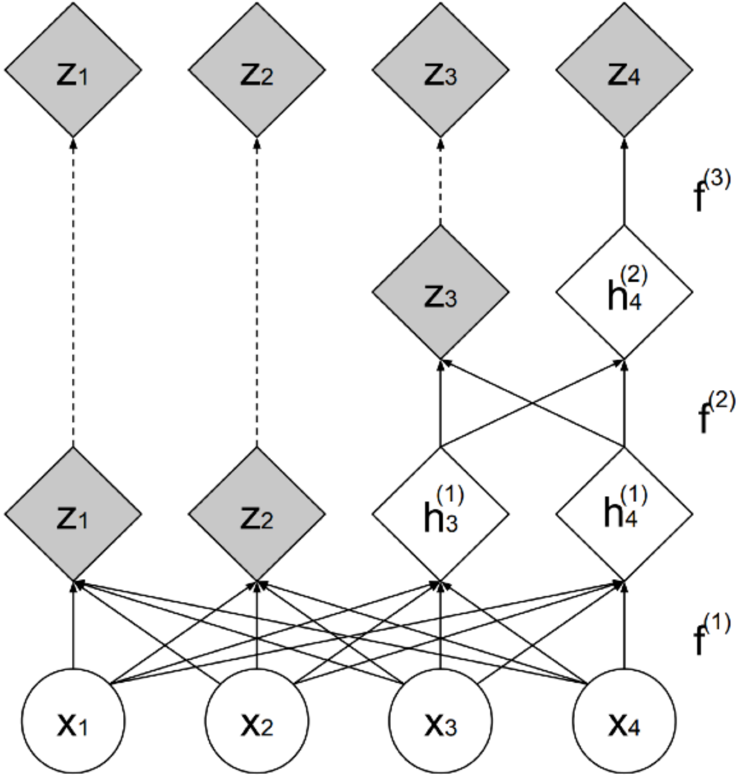
What about the Jacobian?

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

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

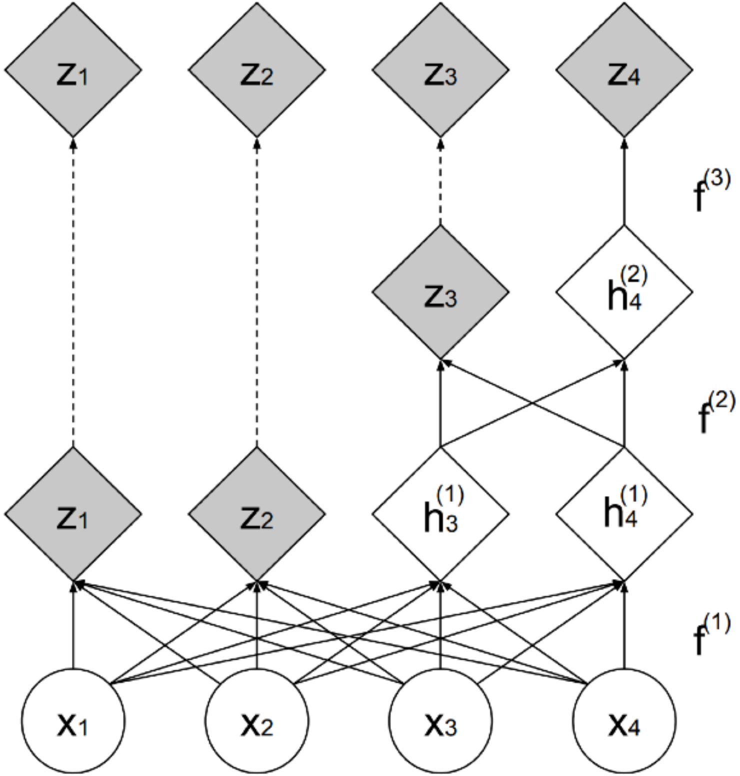
Easy to calculate!

We can also introduce the idea of autoregressive modeling here as well:



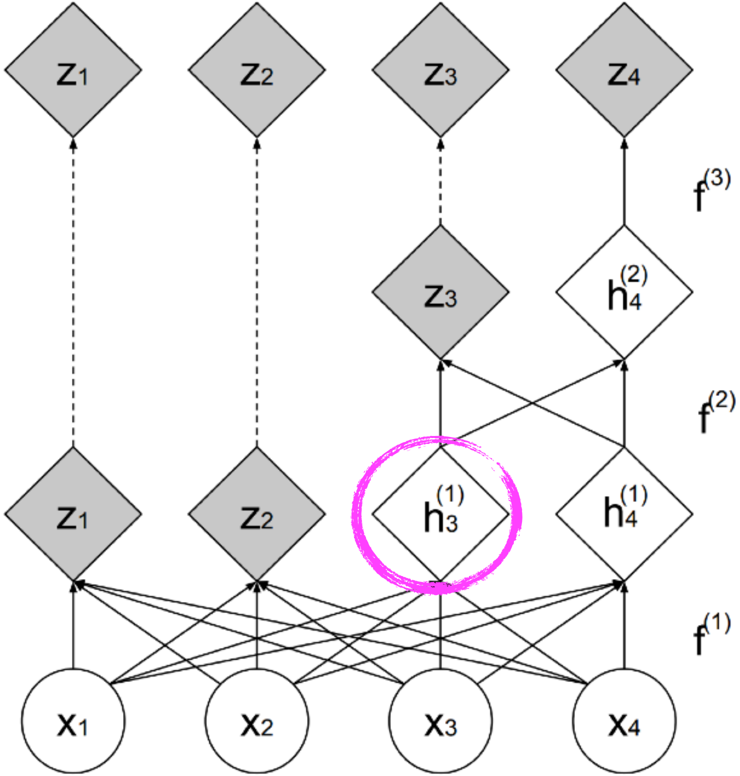
We can also introduce the idea of autoregressive modeling here as well:

$$p(z_1)p(z_2)p(z_3 \,|\, z_1, z_2)p(z_4 \,|\, z_1, z_2, z_3)$$



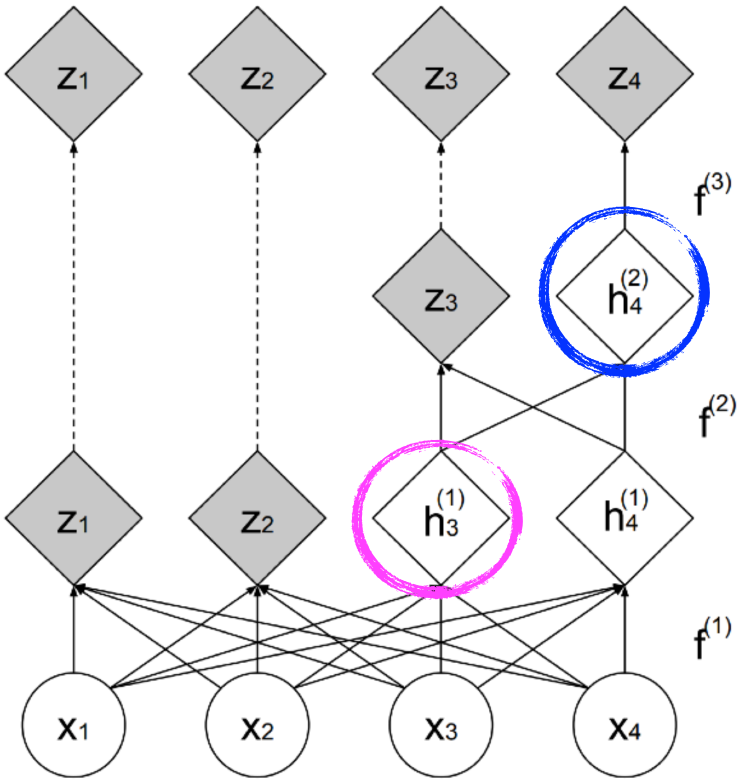
We can also introduce the idea of autoregressive modeling here as well:

$$p(z_1)p(z_2)p(z_3 \mid z_1, z_2)p(z_4 \mid z_1, z_2, z_3)$$



We can also introduce the idea of autoregressive modeling here as well:

$$p(z_1)p(z_2)p(z_3 \mid z_1, z_2)p(z_4 \mid z_1, z_2, z_3)$$



Moreover, we can use additional transformations:

1. **Permutations** of variables (this is invertible).

→ this helps to "mix" variables.

2. Divide variables using a **checkerboard pattern**.

→ this helps to learn higher-order dependencies.

3. Use ***squeezing***: reshape input tensor

→ reshaping can help to "mix" variables.

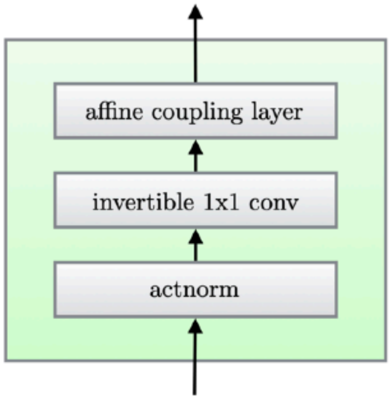
REALNVP

REALNVP

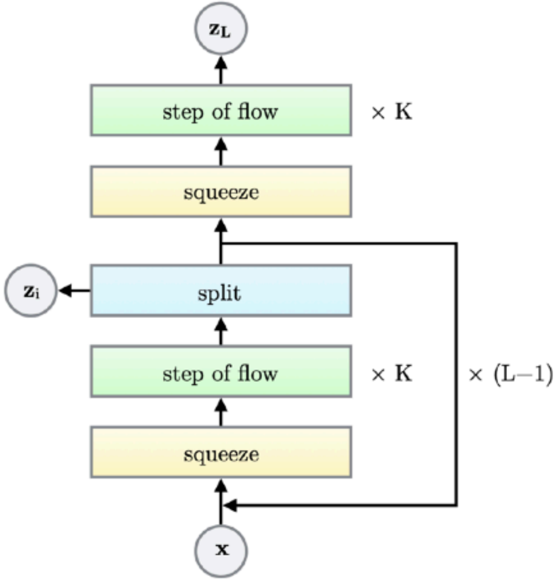
GLOW: REALNVP WITH 1X1 CONVOLUTIONS

A model contains ~1000 convolutions.

A new component: 1x1 convolution instead of a permutation matrix.



(a) One step of our flow.



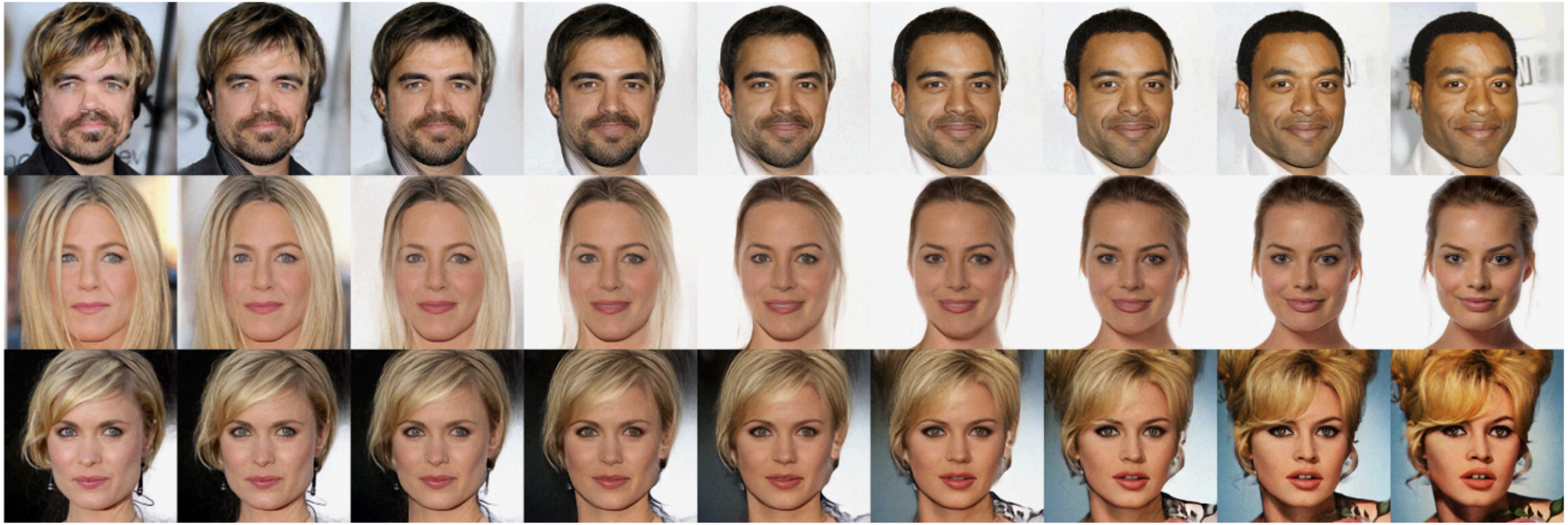
(b) Multi-scale architecture (Dinh et al., 2016).

Kingma, D. P., & Dhariwal, P. (2018). **GLOW**: Generative flow with invertible 1x1 convolutions. *NeurIPS 2018*

GLOW: SAMPLES

CelebAHQ

GLOW: LATENT INTERPOLATION



CelebAHQ

VAES WITH NORMALIZING FLOWS

$$q(\mathbf{z}|\mathbf{x}) \tilde{\propto} p(\mathbf{x}|\mathbf{z})\ p(\mathbf{z})$$

Variational inference with normalizing flows

Rezende & Mohamed. "Variational inference with normalizing flows."

v.d. Berg, Hasenclever, Tomczak, Welling, “Sylvester normalizing flows for variational inference”

Kingma, Salimans, Jozefowicz, Chen, Sutskever, Welling “Improved variational inference with inverse autoregressive flow”

Tomczak, Welling, “Improving variational auto-encoders using householder flow”

Flow-based priors

Chen, Kingma, Salimans, Duan, Dhariwal, Schulman, Abbeel, “Variational lossy autoencoder”

Gatopoulos, Tomczak. "Self-Supervised Variational Auto-Encoders."

Thank you!