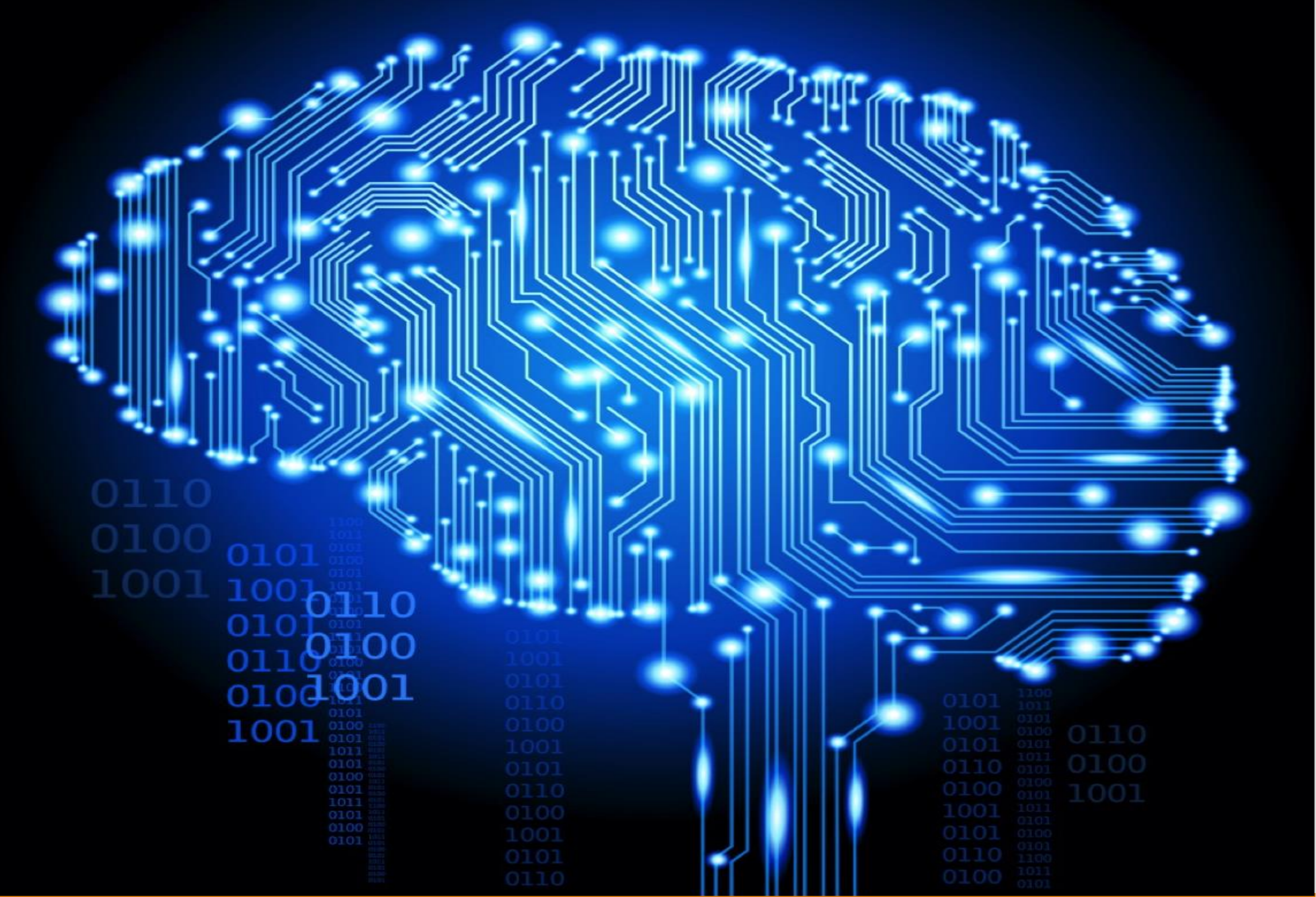


Lecture 11: Advanced Generative Models

Efstratios Gavves

Lecture overview

- Exact likelihood models
  - Autoregressive Models
  - Non-autoregressive flow-based models
- Autoregressive Models
  - NADE, MADE, PixelCNN, PixelCNN++, PixelRNN
- Normalizing Flows
- Non-autoregressive flow-based models
  - RealNVP
  - Glow
  - Flow++

Autoregressive Models

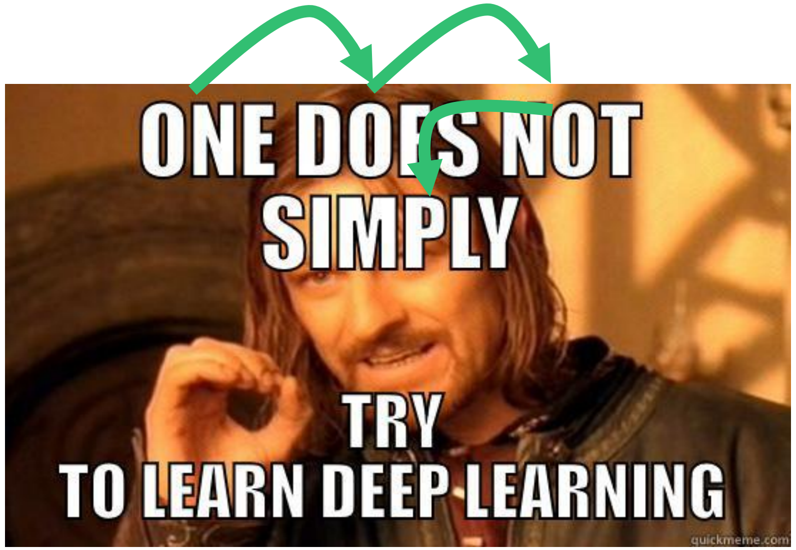
- Let’s assume we have signal modelled by an input random variable x
  - Can be an image, video, text, music, temperature measurements

- Is there an order in all these signals?

Autoregressive Models

- Let's assume we have signal modelled by an input random variable x
  - Can be an image, video, text, music, temperature measurements

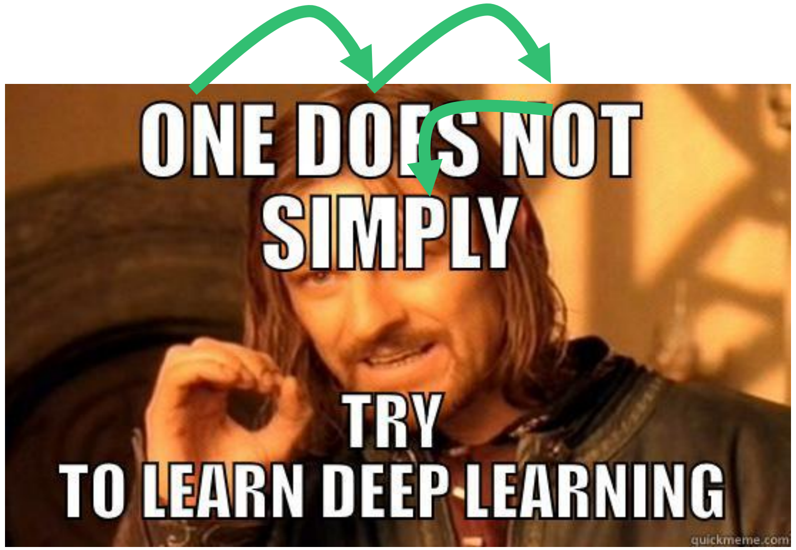
- Is there an order in all these signals?



Autoregressive Models

- Let’s assume we have signal modelled by an input random variable x
  - Can be an image, video, text, music, temperature measurements

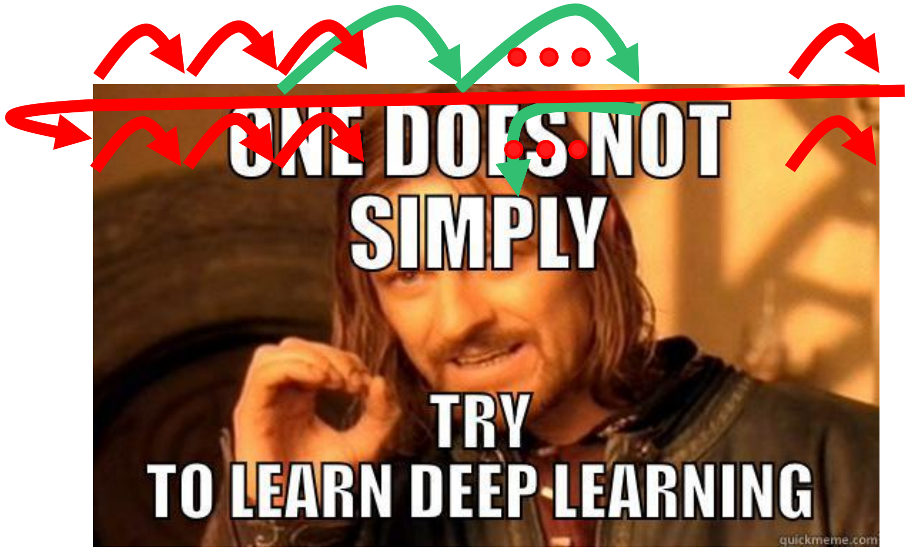
- Is there an order in all these signals? Other signals and orders?



Autoregressive Models

- Let's assume we have signal modelled by an input random variable x
  - Can be an image, video, text, music, temperature measurements

- Is there an order in all these signals?



Autoregressive Models

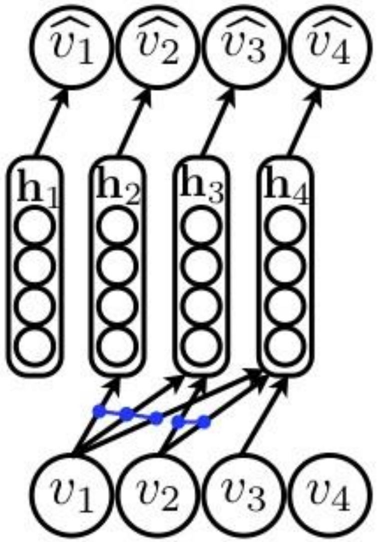
- If x is sequential, there is an order: $x = [x_1, \dots, x_k]$
  - E.g., the order of words in a sentence
- If x is *not* sequential, we can create an artificial order $x = [x_{r(1)}, \dots, x_{r(k)}]$
  - E.g., the order with which pixels make (generate) an image
- Then, the marginal likelihood is a product of conditionals

$$p(x) = \prod_{k=1}^{D} p(x_k | x_{j<k})$$

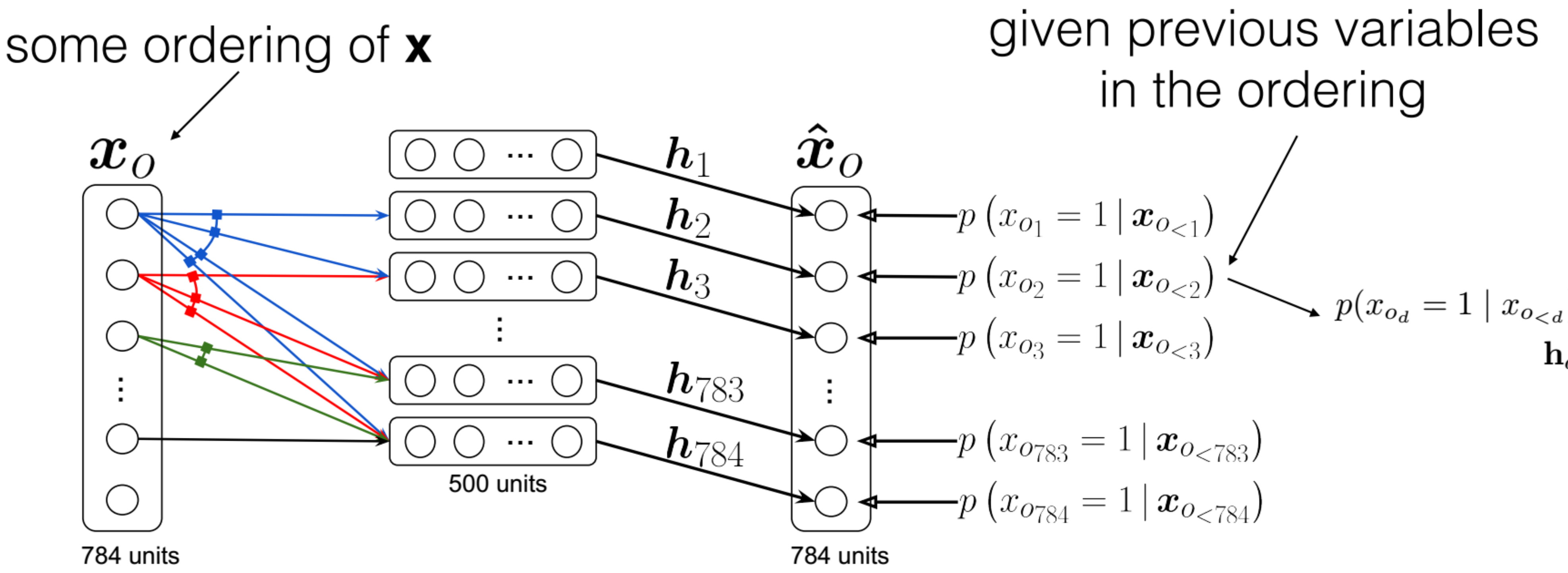
- Different from Recurrent Neural Networks
  (a) no parameter sharing
  (b) chains are not infinite in length

Autoregressive Models

- Pros: because of the product decomposition, $p(x)$ is tractable
- Cons: because the $p(x)$ is sequential, training is slower
  - To generate every new word/frame/pixel, the previous words/frames/pixels in the order must be generated first → no parallelism



NADE



$$p(x_{o_d} = 1 \mid x_{o_{<d}}) = \sigma(V_{o_d,:}\mathbf{h}_d + b_{o_d})$$
$$\mathbf{h}_d = \sigma(W_{:,o_d}\mathbf{x}_{o_d} + \mathbf{c})$$

Neural Autoregressive Distribution Estimation, Larochelle and Murray, AISTATS 2011

NADE

- Minimizing negative log-likelihood as usual

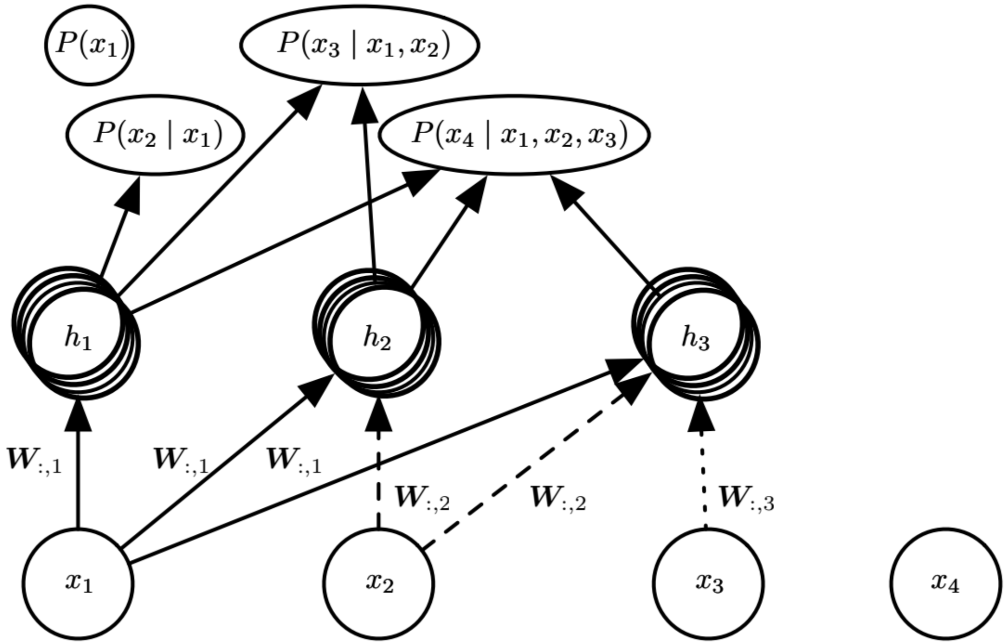
$$\mathcal{L} = -\log p(x) = -\sum_{k=1}^{D} p(x_k|x_{<k})$$

- Then, we model the conditional as

$$p(x_d|x_{<d}) = \sigma(\mathrm{V}_{\mathrm{d},:} \cdot h_d + \mathrm{b}_\mathrm{d})$$

where the latent variable h_d is defined as

$$h_d = \sigma(W_{:,<d} \cdot x_{<d} + c)$$

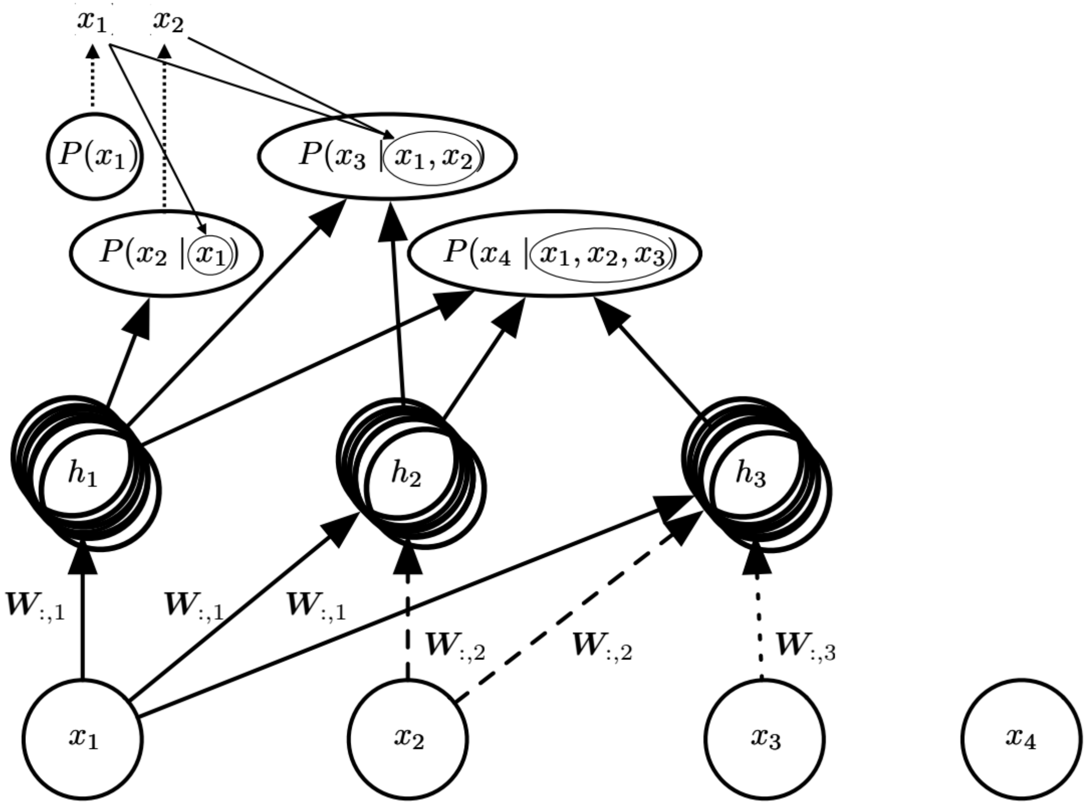


Where W is shared between conditionals

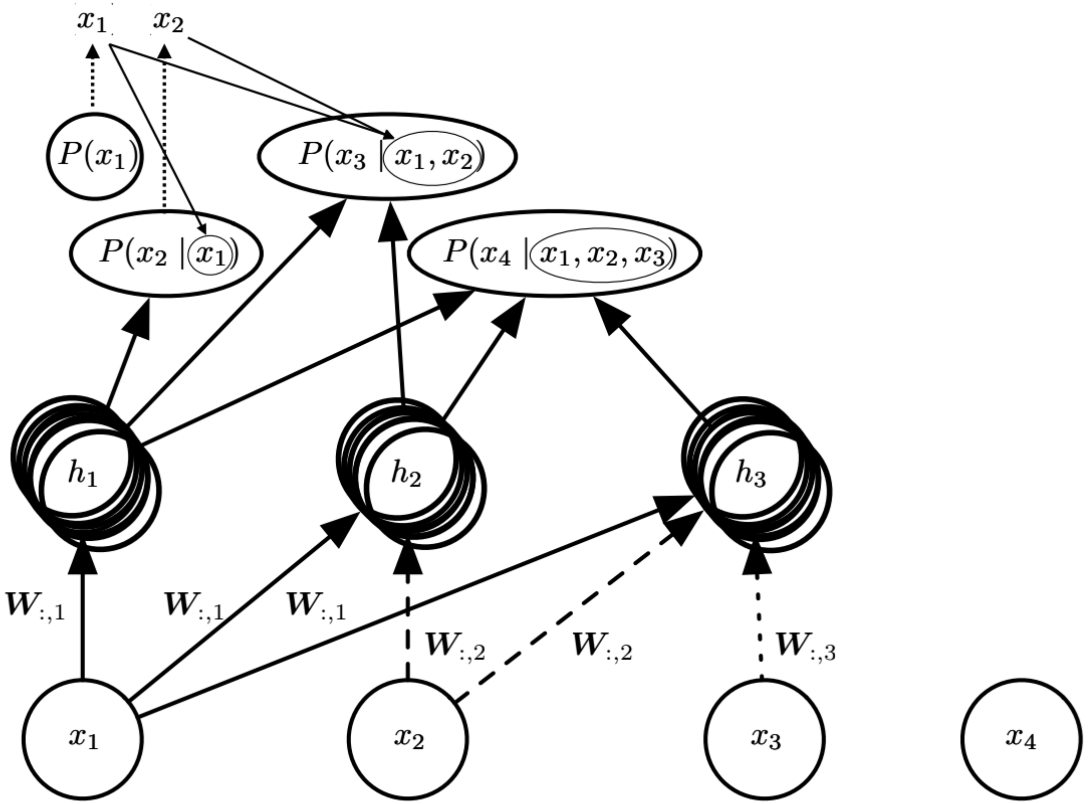
NADE: Training & Testing

- “Teacher forcing” training

Training: Use ground truth values (e.g. of pixels)



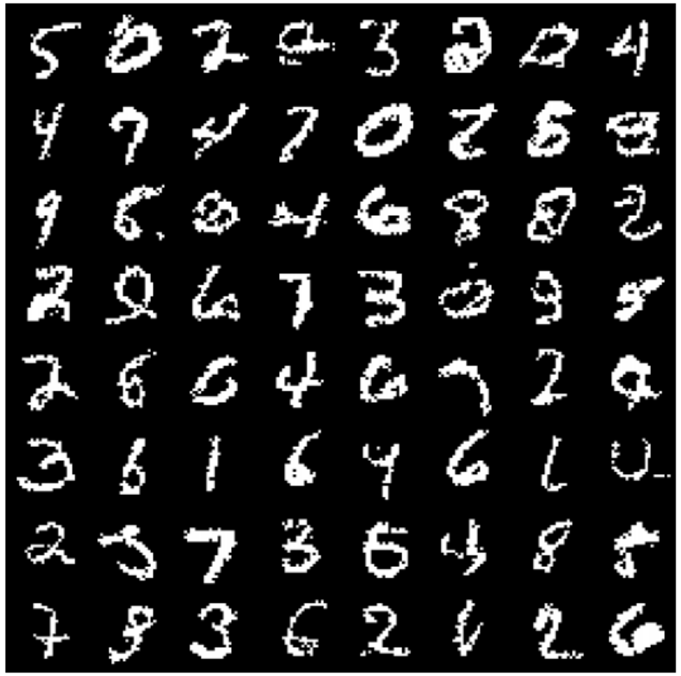
Testing: Use predicted values in previous order



NADE Visualizations

Binarized MNIST samples (NADE)

Binarized MNIST samples (DeepNADE)



Binarized MNIST samples (ConvNADE)

MADE

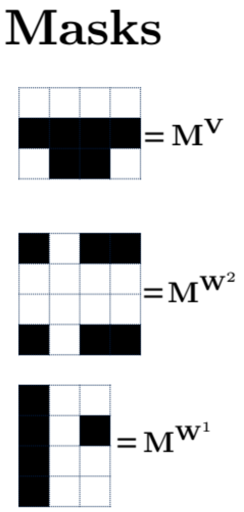
- **Question:** How could we construct an autoregressive autoencoder?
- **To rephrase:** How to modify an autoencoder such that each output x_k depends <u>**only**</u> on the previous outputs $x_{<k}$ (autoregressive property)?
  - Namely, the <u>**present**</u> k-th output $\tilde{x}_k$ must not depend on a computational path from future inputs $x_{k+1}, \ldots, x_D$
  - Autoregressive: $p(x|\theta) = \prod_{k=1}^{D} p(x_k|x_{j<k}, \theta)$
  - Autoencoder: $p(\tilde{x}|x, \theta) = \prod_{k=1}^{D} p(\tilde{x}_k|x_k, \theta)$

Masked Autoencoder for Distribution Estimation, Germain, Mathieu et al., ICML 2015

MADE

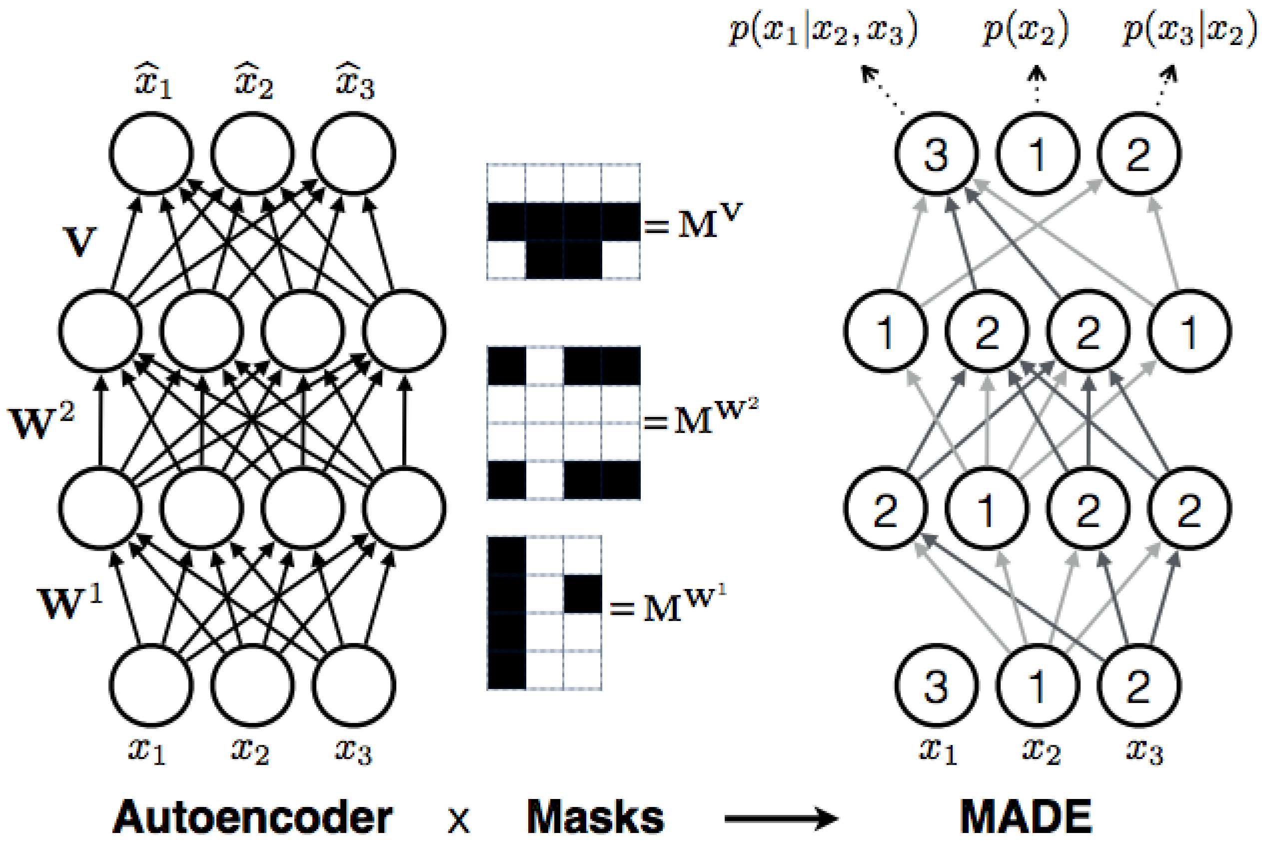
- **Question:** How could we construct an autoregressive autoencoder?
- **To rephrase:** How to modify an autoencoder such that each output x_k depends <u>only</u> on the previous outputs $x_{<k}$ (autoregressive property)?
  - Namely, the <u>present</u> k-th output $\tilde{x}_k$ must not depend on a computational path from future inputs $x_{k+1}, \dots, x_D$
  - Autoregressive: $p(x|\theta) = \prod_{k=1}^{D} p(x_k|x_{j<k}, \theta)$
  - Autoencoder: $p(\tilde{x}|x, \theta) = \prod_{k=1}^{D} p(\tilde{x}_k|x_k, \theta)$
- Answer: Masked convolutions!

$$h(x) = g(b + (W \odot M^W) \cdot x)$$
$$\tilde{x} = \sigma(c + (V \odot M^V) \cdot h(x))$$



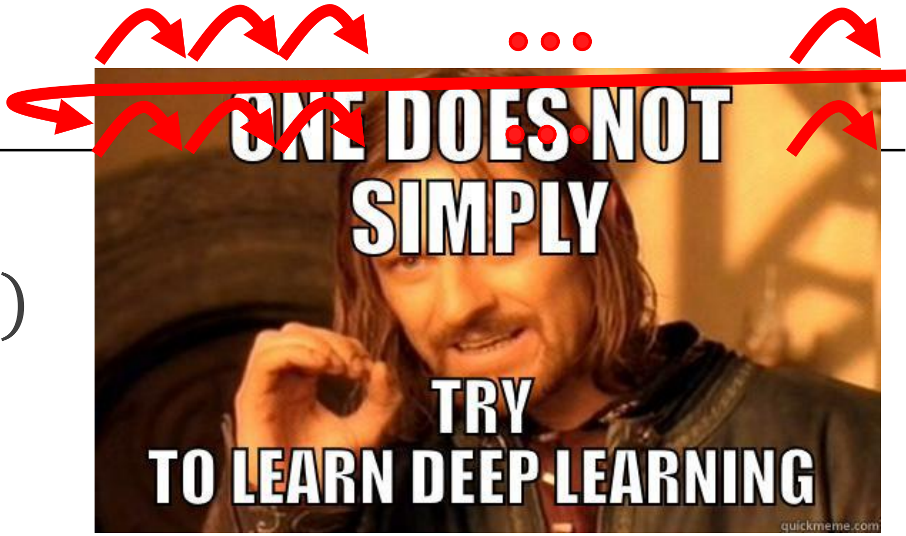
Masked Autoencoder for Distribution Estimation, Germain, Mathieu et al., ICML 2015

MADE



Masked Autoencoder for Distribution Estimation, Germain, Mathieu et al., ICML 2015

PixelRNN



- Unsupervised learning: learn how to model $p(x)$
- Decompose the marginal

$$p(x) = \prod_{i=1}^{n^2} p(x_i | x_1, \dots, x_{i-1})$$

- Assume row-wise pixel by pixel generation and sequential colors R→G→B
  - Each color conditioned on all colors from previous pixels and specific colors in the same pixe

$$p(x_{i,R} | x_{<i}) \cdot p(x_{i,G} | x_{<i}, x_{i,R}) \cdot p(x_{i,B} | x_{<i}, x_{i,R}, x_{i,G})$$

- Final output is 256-way softmax

Pixel Recurrent Neural Networks, van den Oord, Kalchbrenner and Kavukcuoglu, arXiv 2016

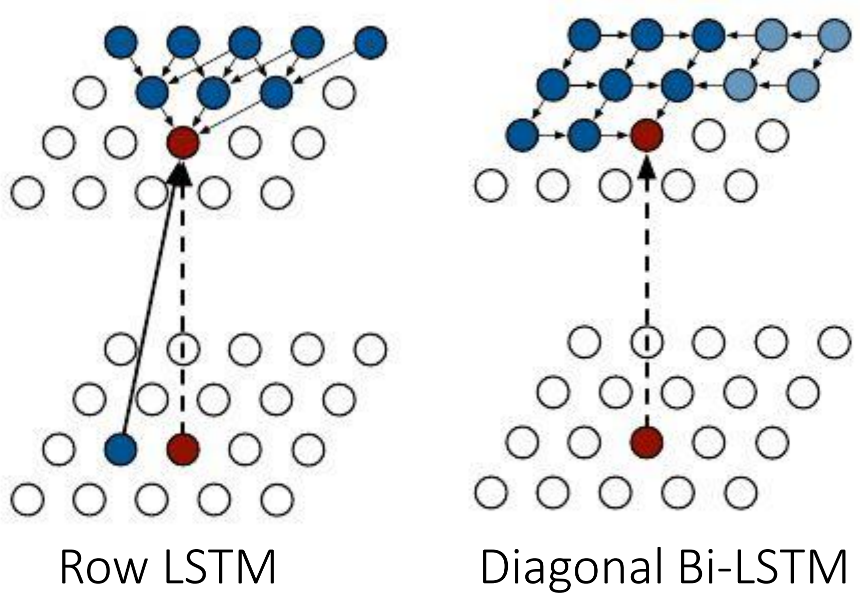
PixelRNN

- How to model the conditionals?

$$p(x_{i,R}|x_{<i}), p(x_{i,G}|x_{<i}, x_{i,R}), p(x_{i,B}|x_{<i}, x_{i,R}, x_{i,G})$$

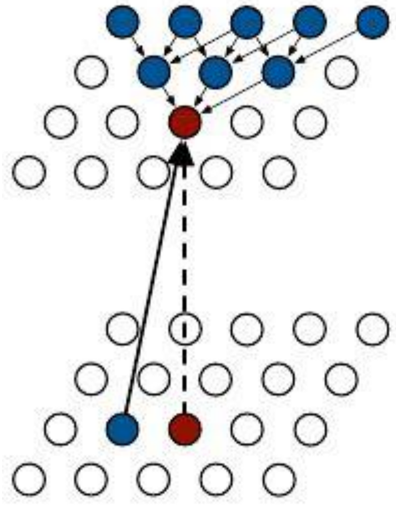
- LSTM variants
  - 12 layers

- Row LSTM
- Diagonal Bi-LSTM



PixelRNN - RowLSTM

- Hidden state (i, j) =
  Hidden state (i-1, j-1) +
  Hidden state (i-1, j) +
  Hidden state (i-1, j+1) +
  p(i, j)
- By recursion the hidden state captures a fairly triangular region

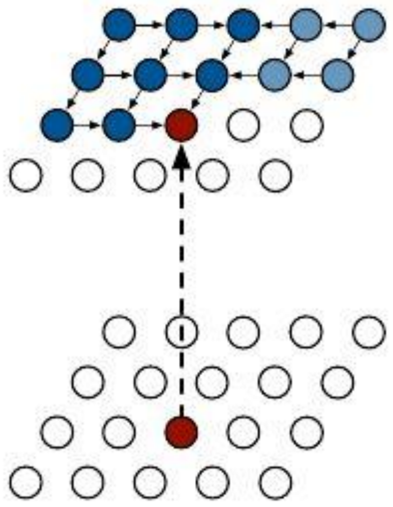


Row LSTM

PixelRNN – Diagonal BiLSTM

- How to capture the whole previous context
- Pixel (i, j) =
  Pixel (i, j-1) +
  Pixel (i-1, j)

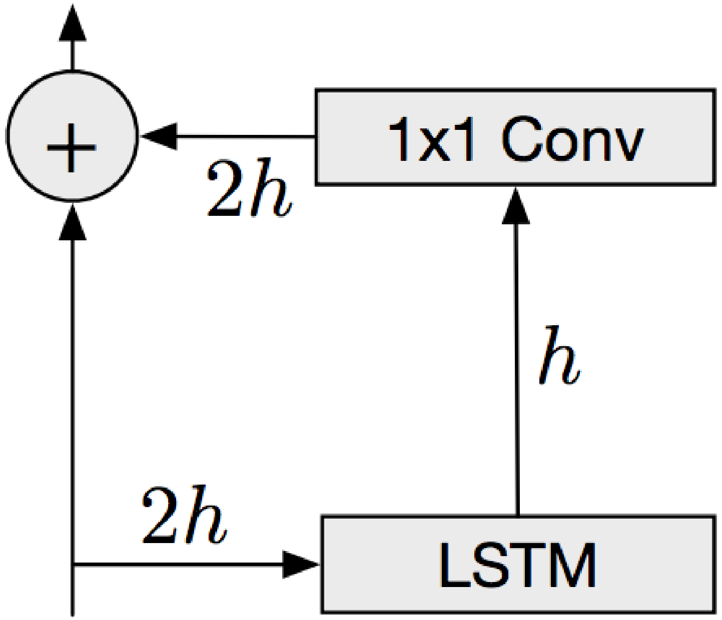
- Processing goes on diagonally
- Receptive layer encompasses entire region



Diagonal Bi-LSTM

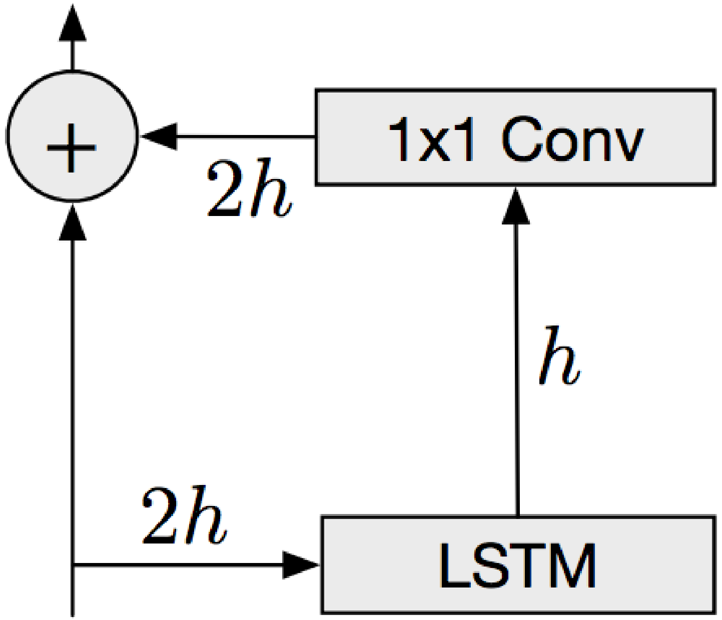
PixelRNN – Residual connections

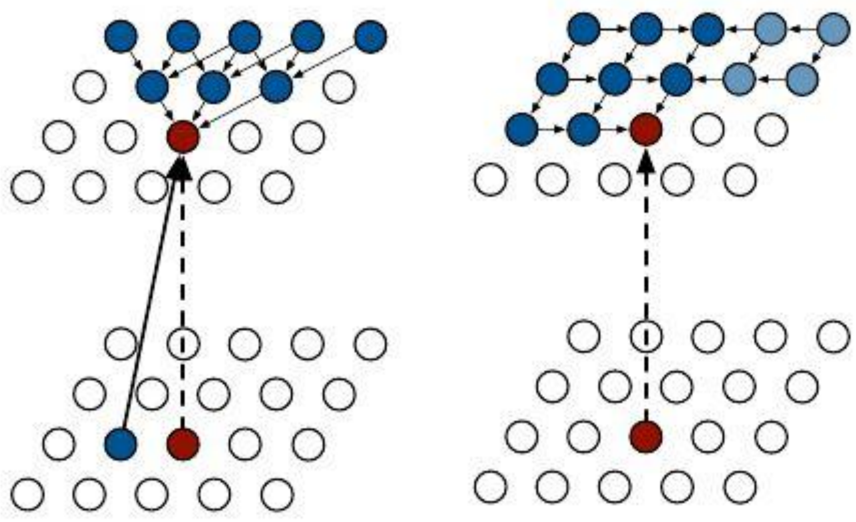
- Propagate signal faster
- Speed up convergence



PixelRNN – Pros & Cons

- Pros: good modelling of $p(x)$ → nice image generation
- Half pro: Residual connections speeds up convergence
- Cons: still slow training, slow generation





Row LSTM

Diagonal Bi-LSTM

PixelRNN - Generations

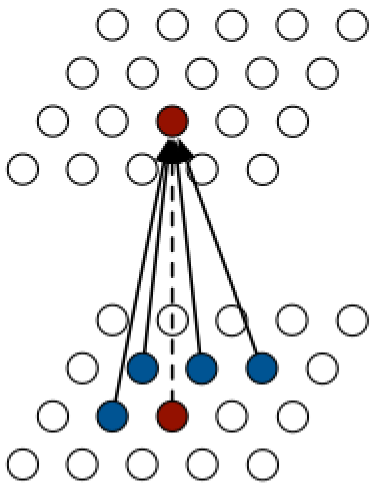
Figure 1. Image completions sampled from a PixelRNN.

PixelCNN

- Unfortunately, PixelRNN is too slow

- Solution: replace recurrent connections with convolutions
  - Multiple convolutional layers to preserve spatial resolution

- Training is much faster because all true pixels are known in advance, so we can parallelize
  - Generation still sequential (pixels must be generated) → still slow



PixelCNN

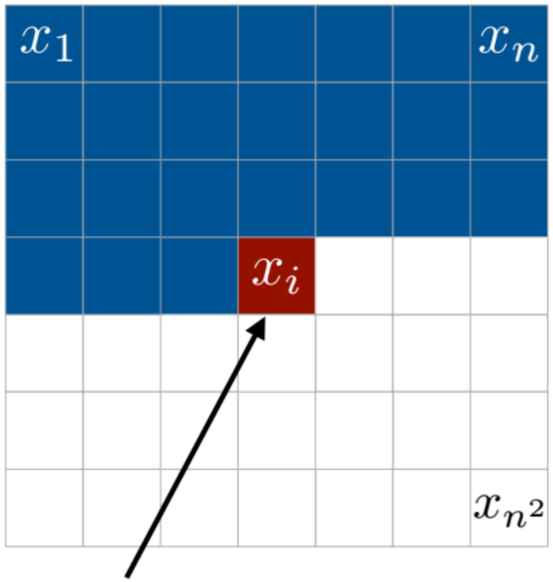
Stack of masked convolutions

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

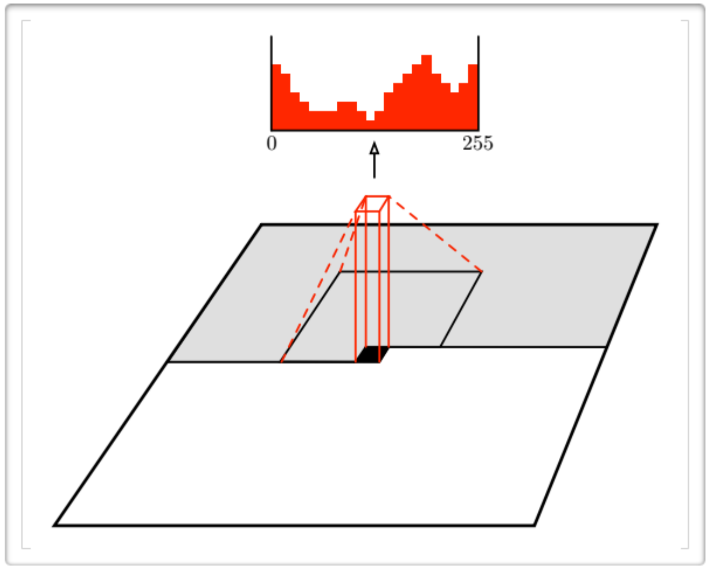
Pixel Recurrent Neural Networks, van den Oord, Kalchbrenner and Kavukcuoglu, arXiv 2016

PixelCNN

- Use masked convolutions again to enforce autoregressive relationships



$$p(x_i \mid \mathbf{x}_{<i})$$



PixelCNN – Pros and Cons

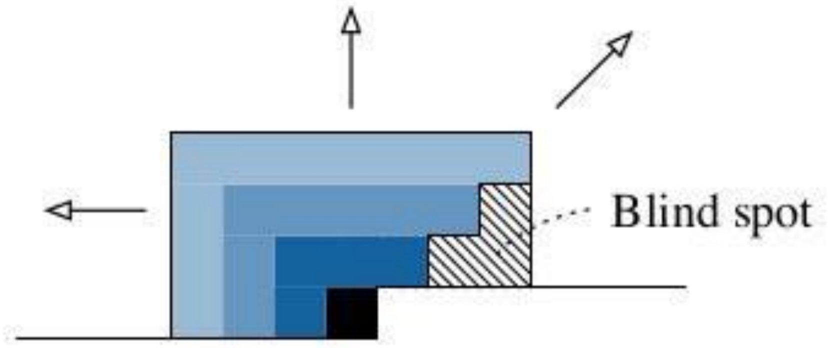
- Cons: Performance is worse than PixelRNN
- Why?

PixelCNN – Pros and Cons

- Cons: Performance is worse than PixelRNN
- Why?
- Not all past context is taken into account
- **New problem:** the cascaded convolutions create a **"blind spot"**

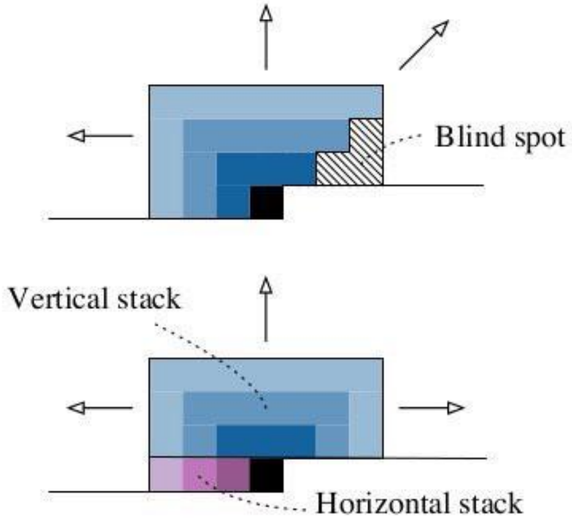
Blind spot

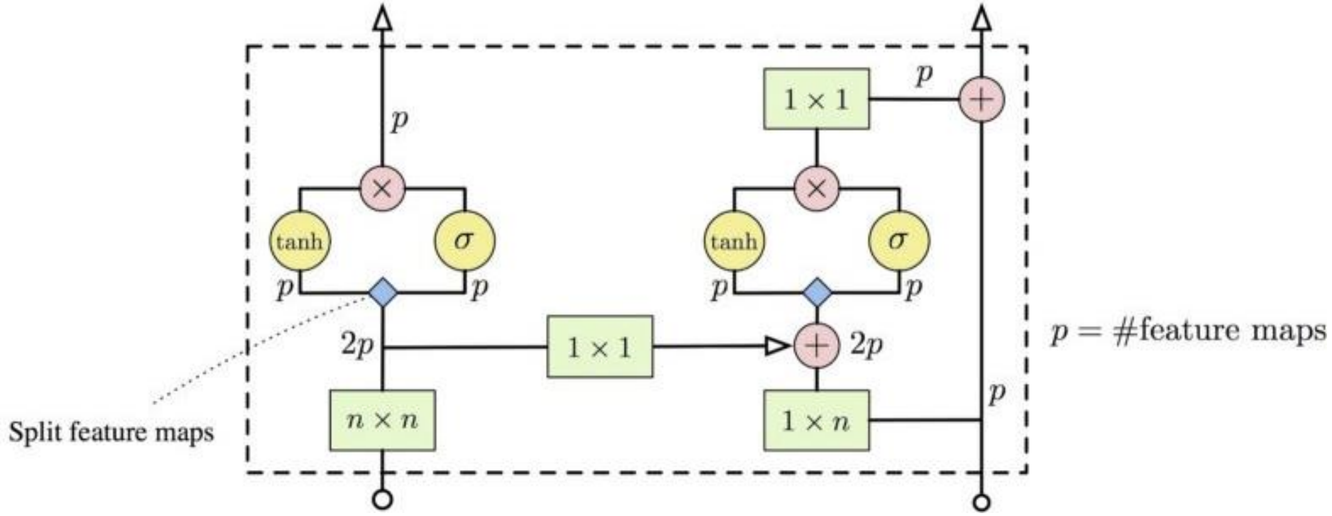
- Because of
  (a) the limited receptive field of convolutions and
  (b) computing all features at once (not sequentially)
  → cascading convolutions makes current pixel not depend on <u>all</u> previous
  → blind spot



Fixing the blind spot: Gated PixelCNN

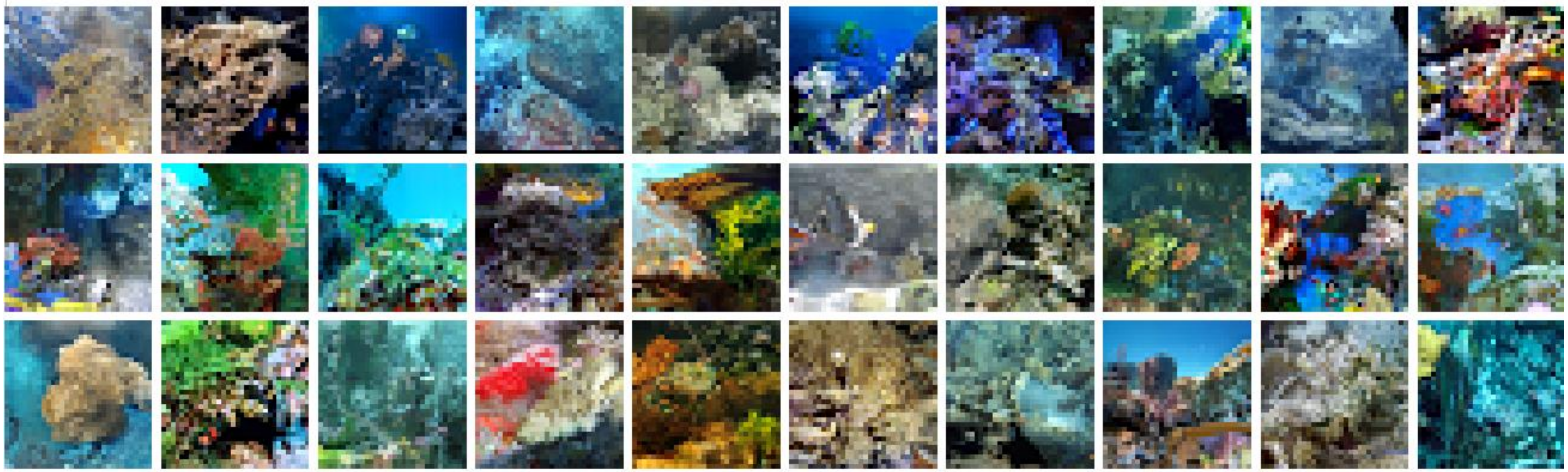
- Use two layers of convolutions stacks
  - Horizontal stack: conditions on current row and takes as input the previous layer output and the vertical stack
  - Vertical stack: conditions on all rows above current pixels
- Also replace ReLU with a $\tanh(W * x) \cdot \sigma(U * x)$





PixelCNN - Generations

- Coral reef



PixelCNN - Generation

- Sorrel horse

PixelCNN - Generation

- Sandbar

PixelCNN - Generation

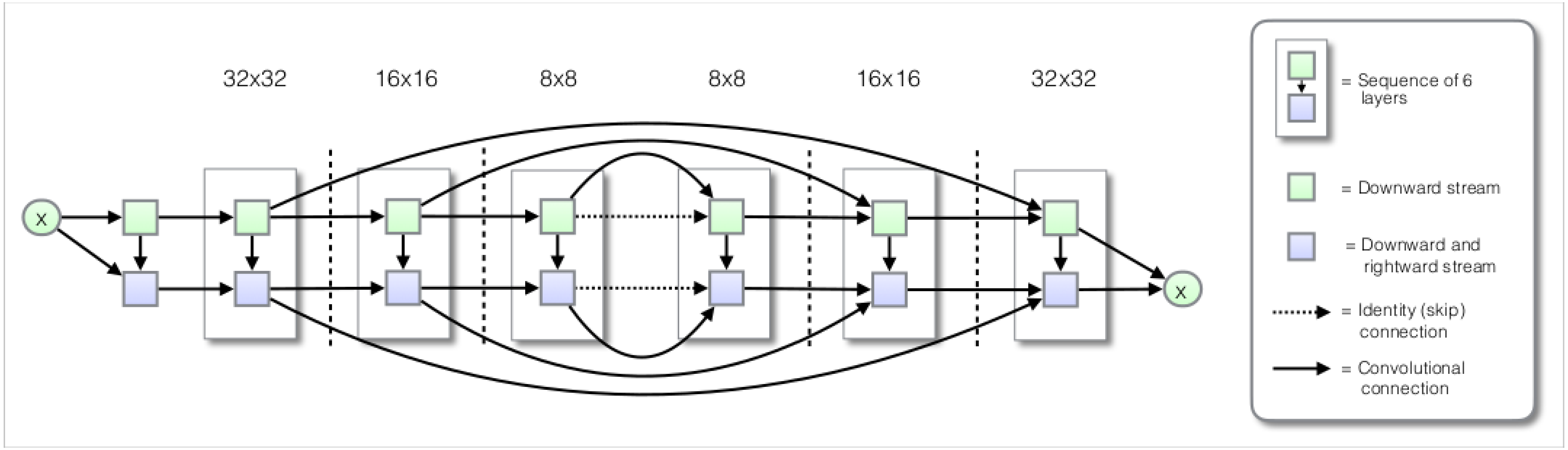
- Lhasa Apso

PixelCNN++

- Improving the PixelCNN model
- Replace the softmax output with a discretized logistic mixture lihelihood
  - Softmax is too memory consuming and gives sparse gradients
  - Instead, assume logistic distribution of intensity and round off to 8-bits
- Condition on whole pixels, not pixel colors
- Downsample with stride-2 convs to compute long-range dependencies
- Use shortcut connections
- Dropout
  - PixelCNN is too powerful a framework → can onverfit easily

PixelCNN++: Improving the PixelCNN with Discretized Logistic, Salimans, Karpathy, Chen, Kingma, ICLR 2017

PixelCNN++



32x32
16x16
8x8
8x8
16x16
32x32
X
X
= Sequence of 6 layers
= Downward stream
= Downward and rightward stream
= Identity (skip) connection
= Convolutional connection

PixelCNN++ - Generations

Advantages/Disadvantages

- SoTA density estimation

- Quite slow because of autoregressive nature
  - They must sample sequentially

- They do not have a latest space

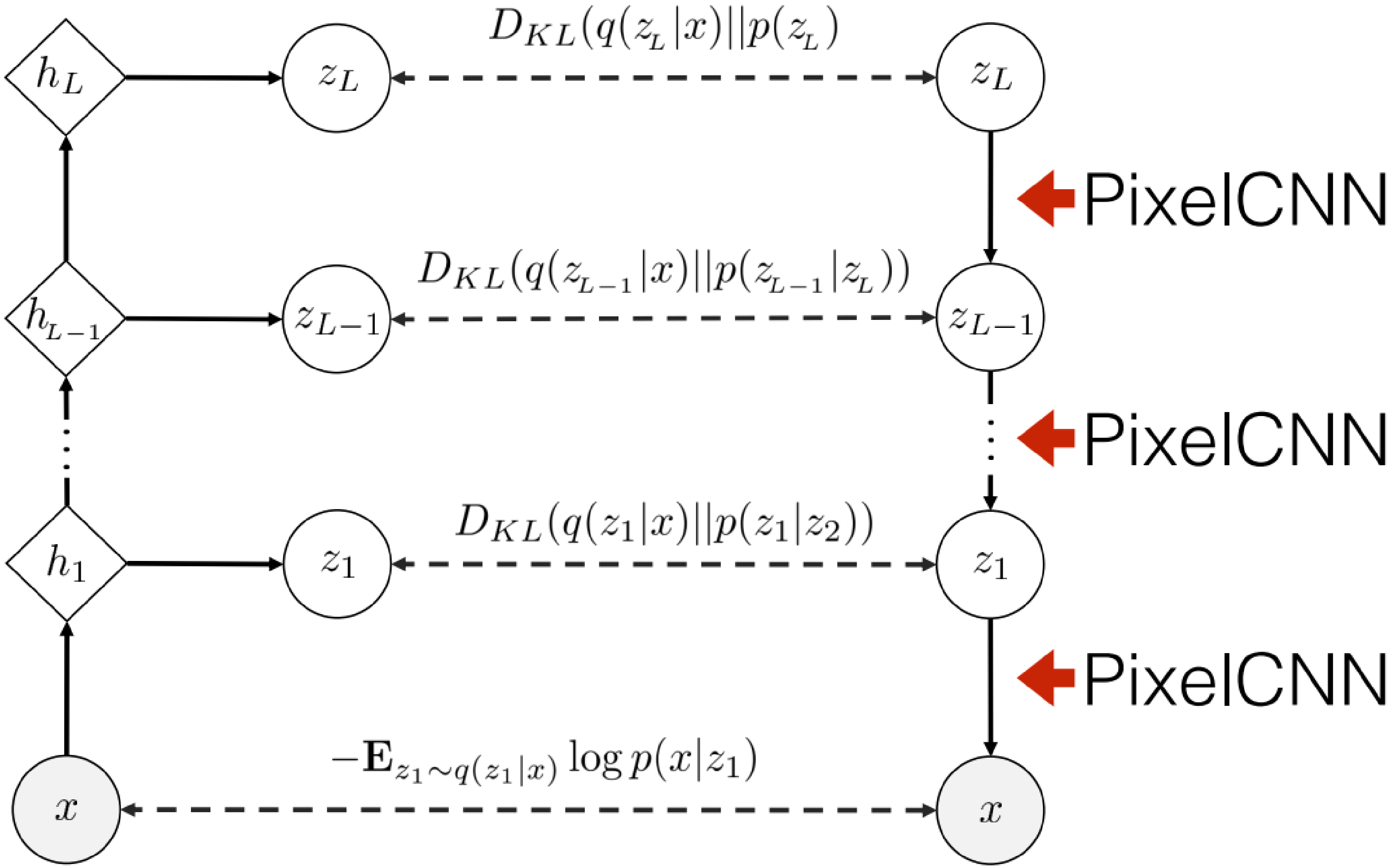
PixelVAE

- A standard VAE with a PixelCNN generator/decoder

- Be careful. Often the generator is so powerful, that the encoder/inference network is ignored ← Whatever the latent code z there will be a nice image generated

PixelVAE: A Latent Variable Model for Natural Images, Gulrajani et al., ICLR 2017

PixelVAE


$D_{KL}(q(z_L|x)||p(z_L))$
$D_{KL}(q(z_{L-1}|x)||p(z_{L-1}|z_L))$
$D_{KL}(q(z_1|x)||p(z_1|z_2))$
$-\mathbf{E}_{z_1 \sim q(z_1|x)} \log p(x|z_1)$
h_L
h_{L-1}
h_1
z_L
z_{L-1}
z_1
x
PixelCNN
PixelCNN
PixelCNN

PixelVAE - Generations

64x64 LSUN Bedrooms

64x64 ImageNet

PixelVAE - Generations

Varying top latents

Varying bottom latents

Varying pixel-level noise

All about Flows

Towards better posteriors

- Ideally, we want to minimize the difference between the approximate posterior and the true posterior

$$\mathrm{ELBO}_{\theta,\varphi}(\mathrm{x}) = \log p_\theta(x) - \mathrm{KL}(q_\varphi(z|x) || \mathrm{p}(\mathrm{z}|\mathrm{x}))$$

- VAEs assume simple diagonal Gaussian priors and posteriors

- Is this a good assumption?
- Could we have better posteriors without blowing up the complexity?

Sampling from a VAE

- Sampling from a Gaussian prior

$$z \sim \mathcal{N}(z | \mu(x), \mathrm{diag}(\sigma^2(x)))$$

Flows: Or, what if we were to transform a simple prior?

- Let's assume that the transformation f is invertible, that is we can compute

$$z_i = f(z_{i-1})$$

as well as

$$z_{i-1} = f^{-1}(z_i)$$

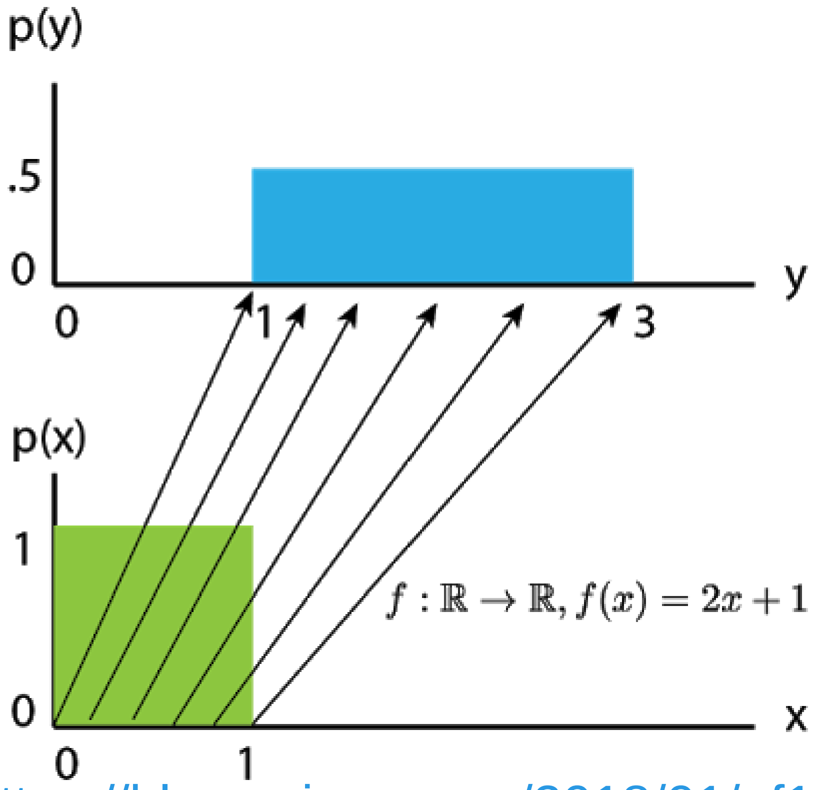
- If z_{i-1} is an RV with a pdf $p_{i-1}(z_{i-1})$, then z_i also an RV
- Easy to compute the pdf of z_i because of the change of variable formula

$$p_i(z_i) = p_{i-1}(f^{-1}(z_i)) \left| \det \frac{df_i^{-1}}{dz_i} \right|$$

And can be shown that $\left| \det \frac{df_i^{-1}}{dz_i} \right| = \left| \det \frac{df_i}{dz_{i-1}} \right|^{-1}$

Change of variable formula

$$p_i(z_i) = p_{i-1}(f^{-1}(z_i)) \left|\det \frac{df_i}{dz_{i-1}}\right|^{-1}$$



https://blog.evjang.com/2018/01/nf1.html

What have we gained?

- We have that

$$p_i(z_i) = p_{i-1}(z_{i-1}) \left|\det \frac{dz_{i-1}}{dz_i}\right|$$

- If we start from a simple pdf, like a Gaussian for $\boldsymbol{z_{i-1}}$
- And we use a transformation $z_i = f(z_{i-1})$ for which
  - It is easy to compute the inverse $z_{i-1} = f^{-1}(z_i)$
  - And is easy to compute the Jacobian $\mathbf{det}\frac{df_i^{-1}}{dz_i}$ and its determinant
- Then by the change of variable we can compute the pdf of z_i
- Even if we do not know the analytic form of $p_i(z_i)$!!

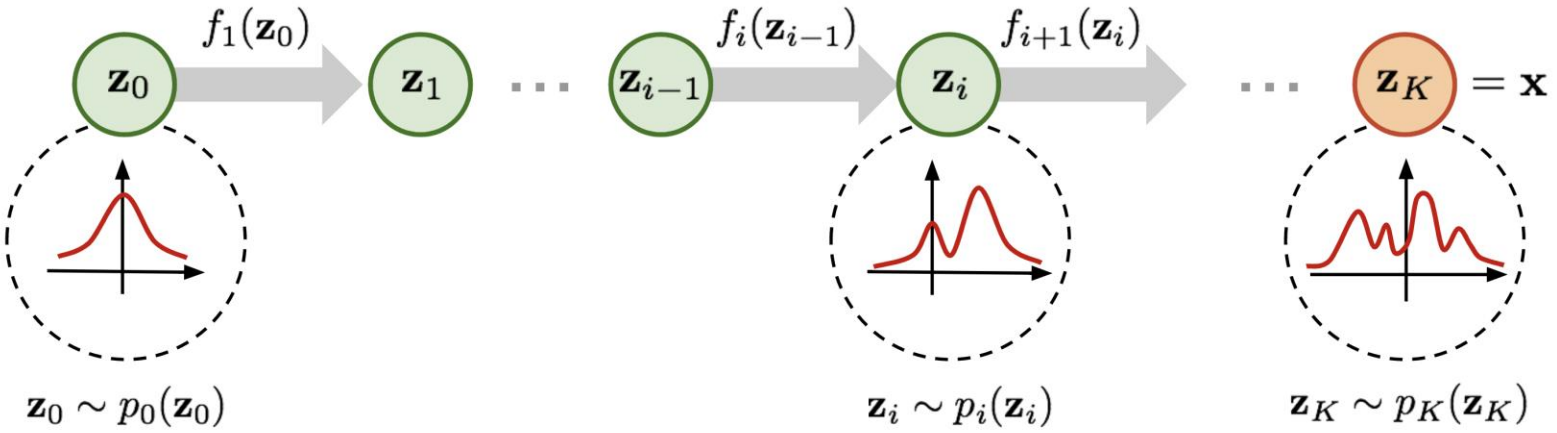
We can even apply this recursively

- If we have a series of K transformations

$$x = z_K = f_K \circ f_{K-1} \circ \cdots \circ f_1(z_0)$$

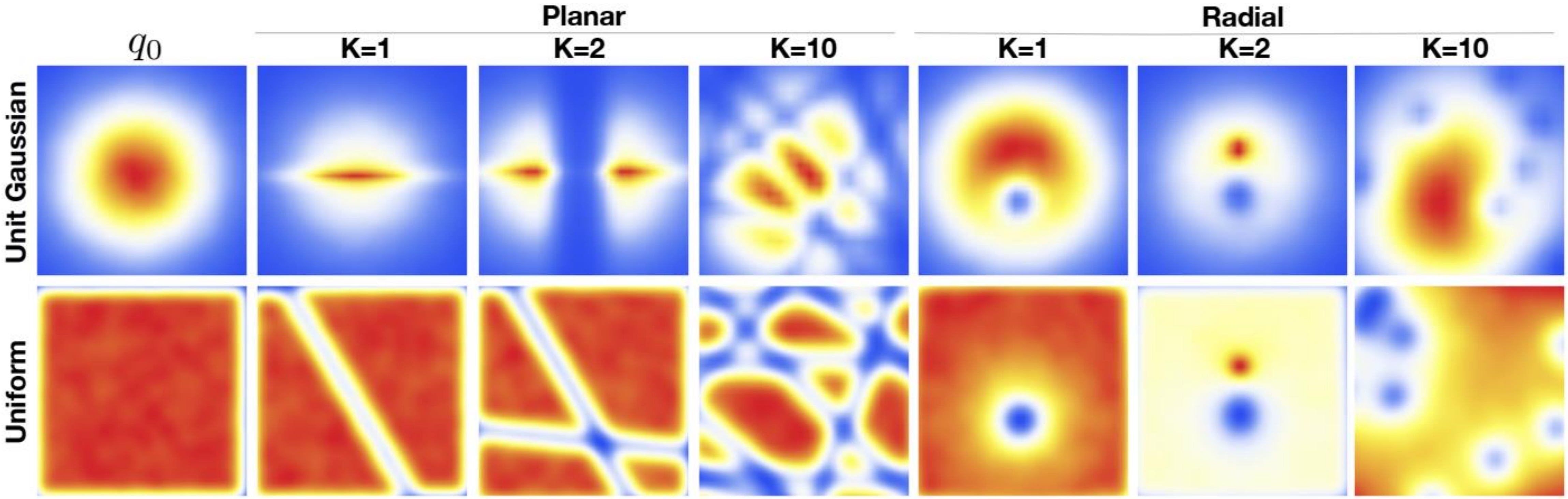
Then by decomposing the log-likelihood we have

$$\log p(x) = \log \pi_0(z_0) - \sum_{i}^{K} \log \left| \det \frac{df_i}{dz_{i-1}} \right|^{-1}$$



https://lilianweng.github.io/lil-log/2018/10/13/flow-based-deep-generative-models.html

How does this look like?



Variational Inference with Normalizing Flows, Rezende and Mohammed, 2015

So, what?

- We can have complex posteriors without really needing to know complex mathematical formulations of the pdf
- Instead, we learn the posteriors from the data
- Hopefully, our posteriors then would be closer to the true posterior

Inference with Flows

- $\text{ELBO}_{\theta,\varphi}(\text{x}) = \mathbb{E}_{z_0 \sim q_0(z_0|x)}[\log p_\theta(x|z_K)]$

$$-KL(q_0(z_0|x)||p_\lambda(z)) + \mathbb{E}_{z_0 \sim q_0(z_0|x)}[\sum_i^K \log \left|\det \frac{df_i}{dz_{i-1}}\right|^{-1}]$$

- The first term is simply the reconstruction term
  - How likely our generations are, after having sampled from our simple $z_0 \sim q_0(z_0|x)$
- The second term is the KL divergence between the flow $q_0(z_0|x)$ and our simple prior $p_\lambda(z)$
- The third term is the accumulation of log determinants from the recursive change of variables

- Although the final posterior q_K will have a complex form, its closed form is not needed for computing any expectations, e.g., in the ELBO
  - Check the Law of Unconscious Statistician (LOTUS)

Pipeline

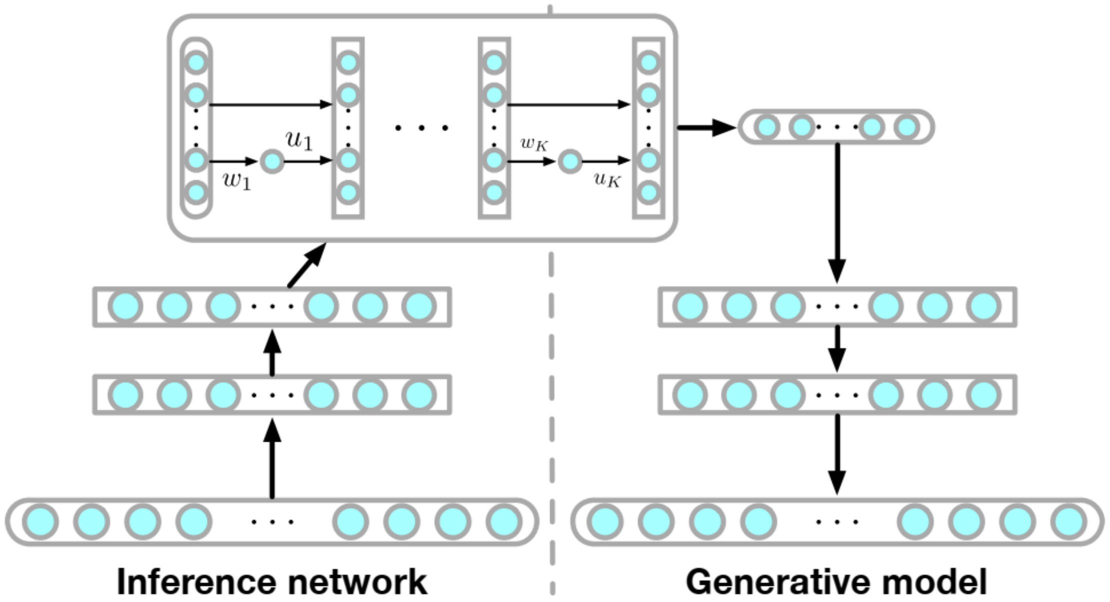


Figure 2. Inference and generative models. Left: Inference network maps the observations to the parameters of the flow; Right: generative model which receives the posterior samples from the inference network during training time. Round containers represent layers of stochastic variables whereas square containers represent deterministic layers.

Variational Inference with Normalizing Flows, Rezende and Mohammed, 2015

Planar flow

- The transformation is

$$f(z) = z + uh(w^T z + b)$$

  - u, w, b are free parameters
  - h is an element-wise non-linearity (element-wise so that it is easy to invert)
  - The log-determinant of the Jacobian is

$$\psi(z) = h'(w^T z + b)w$$

$$\det \left| \frac{\partial f}{\partial z} \right| = |1 + u^T \psi(z)|$$

Radial flow

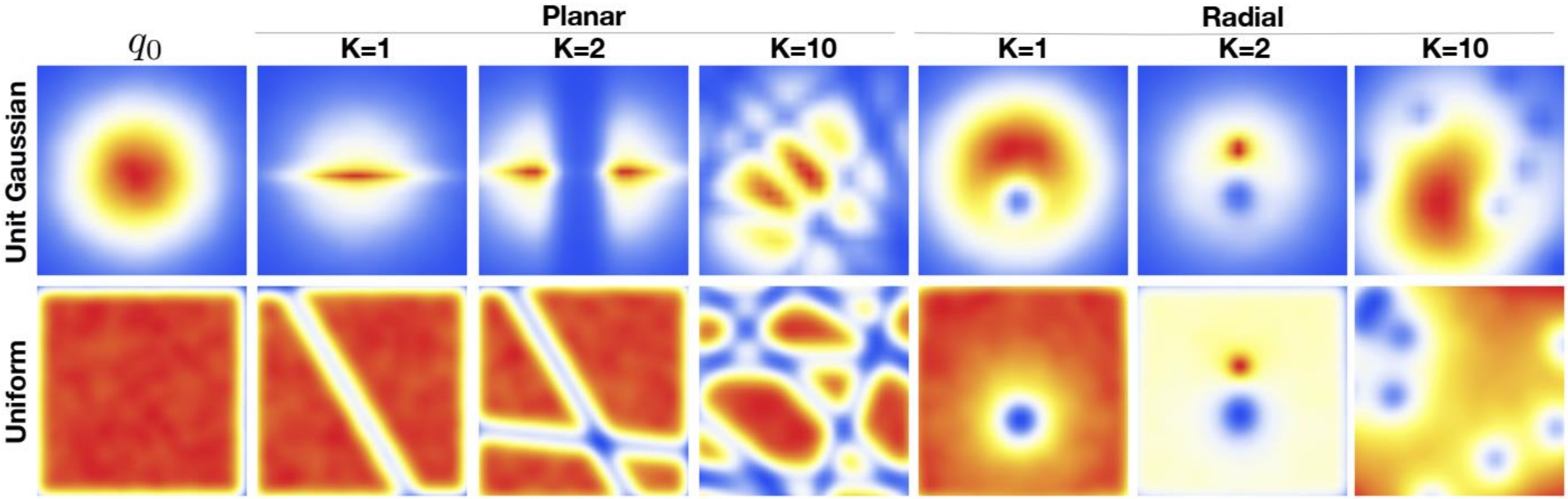
- The transformation is

$$f(z) = z + \beta h(\alpha, r)(z - z_0)$$

Where $h(\alpha, r) = 1/(\alpha + r)$

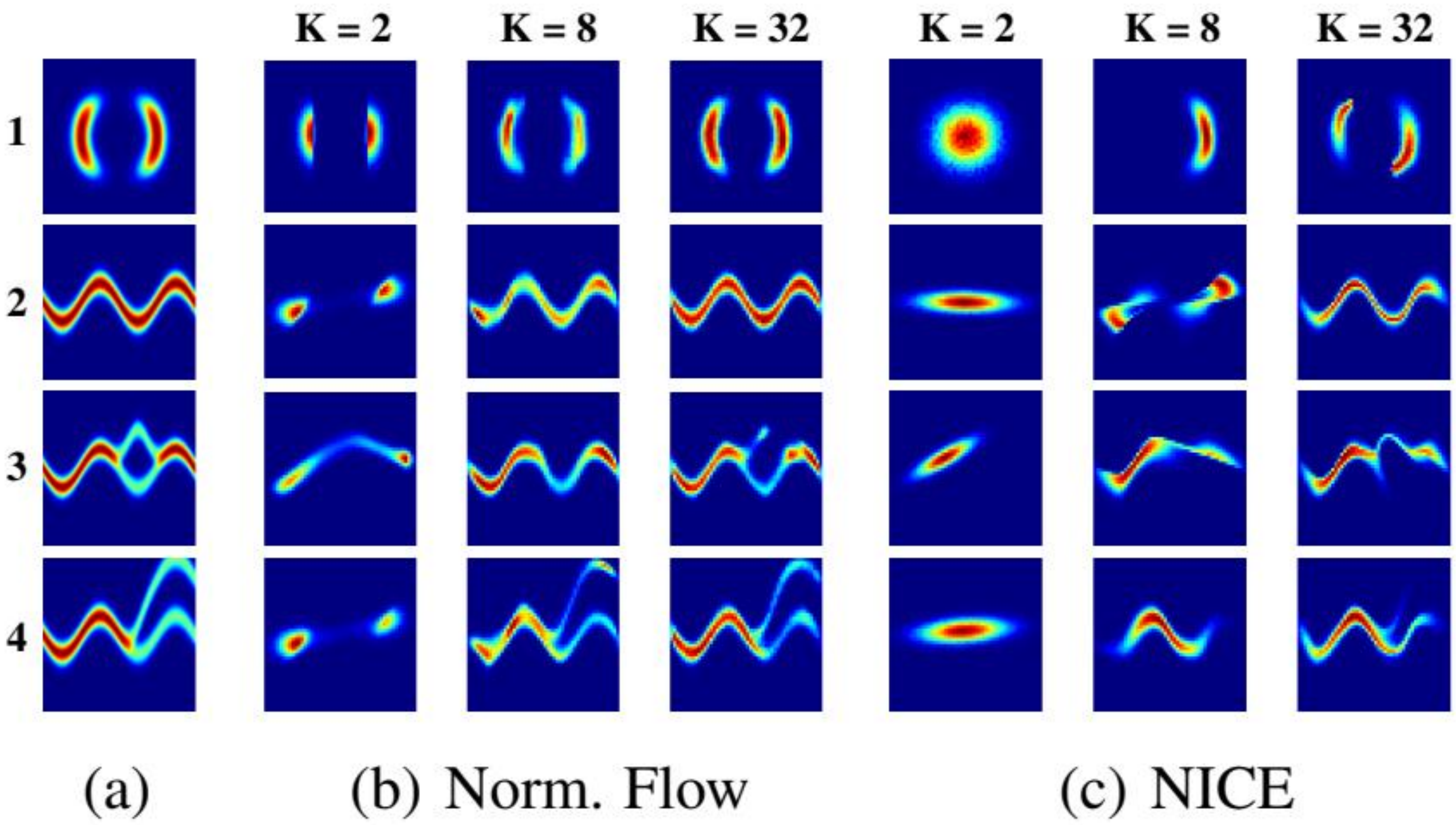
- The log-determinant of the Jacobian is
- $\det\left|\frac{\partial f}{\partial z}\right| = [1 + \beta h(\alpha, r)]^{d-1}[1 + \beta h(\alpha, r) + h'(\alpha, r)r]$

How does this look like?



Variational Inference with Normalizing Flows, Rezend and Mohammed, 2015

Some results



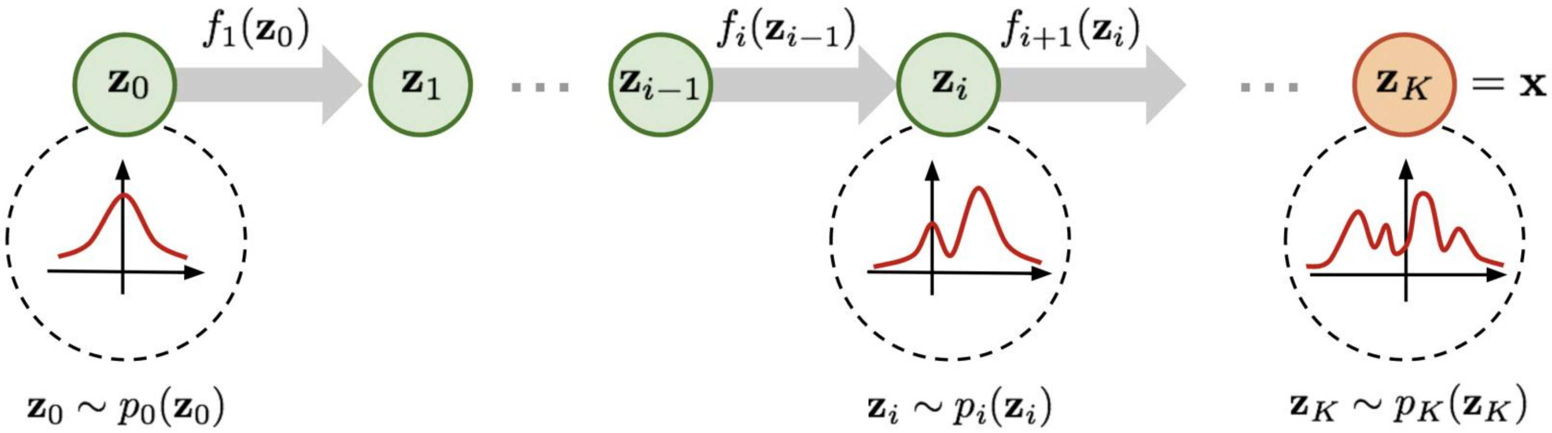
Flow-based models

- Using flows and change of variable formula to compute the exact likelihood $p(x)$ tractably

- Real NVP/NICE
- GLOW
- FLOW++

Real NVP

- Takes the idea of changing variables formula to define an invertible generative model
  - The function f is the encoder/inference network
  - The inverse function f^{-1} is the decoder/generator network
- RealNVP defines computationally efficient operations to scale



A visualization

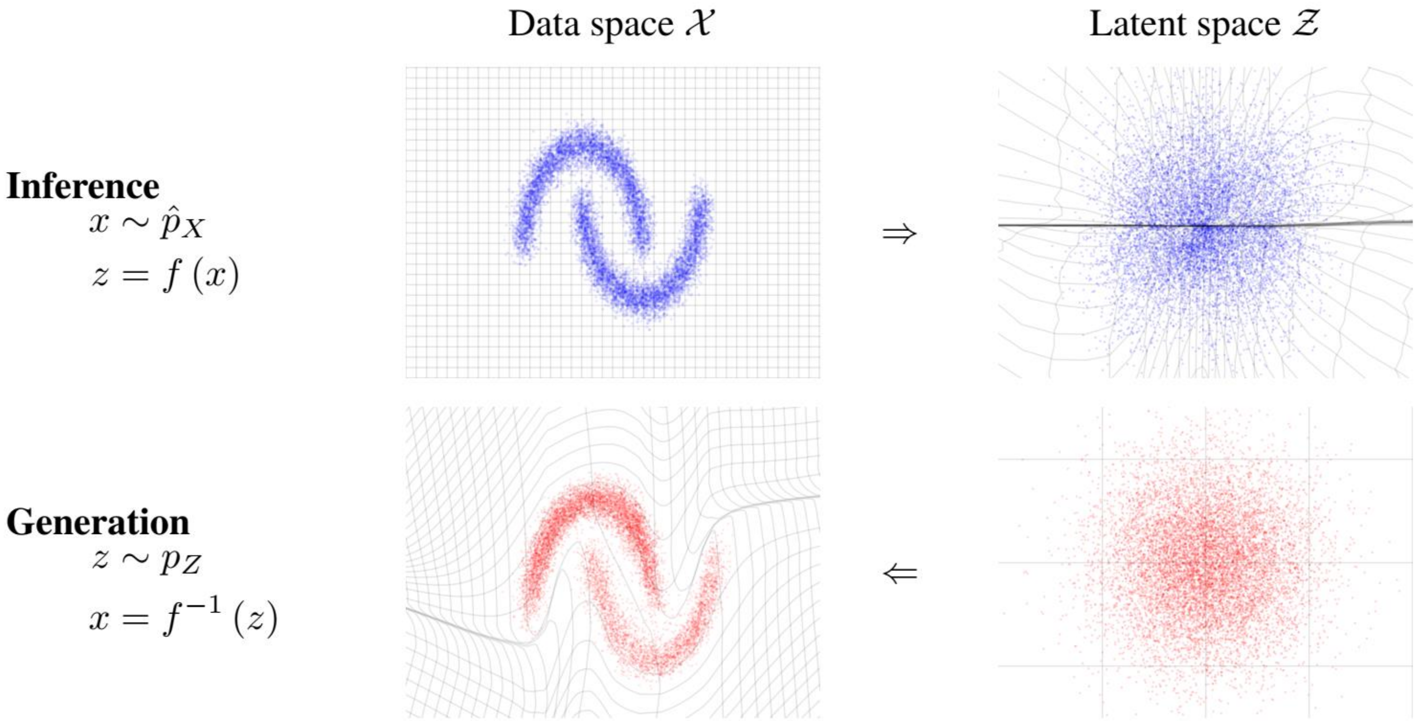


Figure 1: Real NVP learns an invertible, stable, mapping between a data distribution $\hat{p}_X$ and a latent distribution p_Z (typically a Gaussian). Here we show a mapping that has been learned on a toy 2-d dataset. The function $f(x)$ maps samples x from the data distribution in the upper left into approximate samples z from the latent distribution, in the upper right. This corresponds to exact inference of the latent state given the data. The inverse function, $f^{-1}(z)$, maps samples z from the latent distribution in the lower right into approximate samples x from the data distribution in the lower left. This corresponds to exact generation of samples from the model. The transformation of grid lines in $\mathcal{X}$ and $\mathcal{Z}$ space is additionally illustrated for both $f(x)$ and $f^{-1}(z)$.

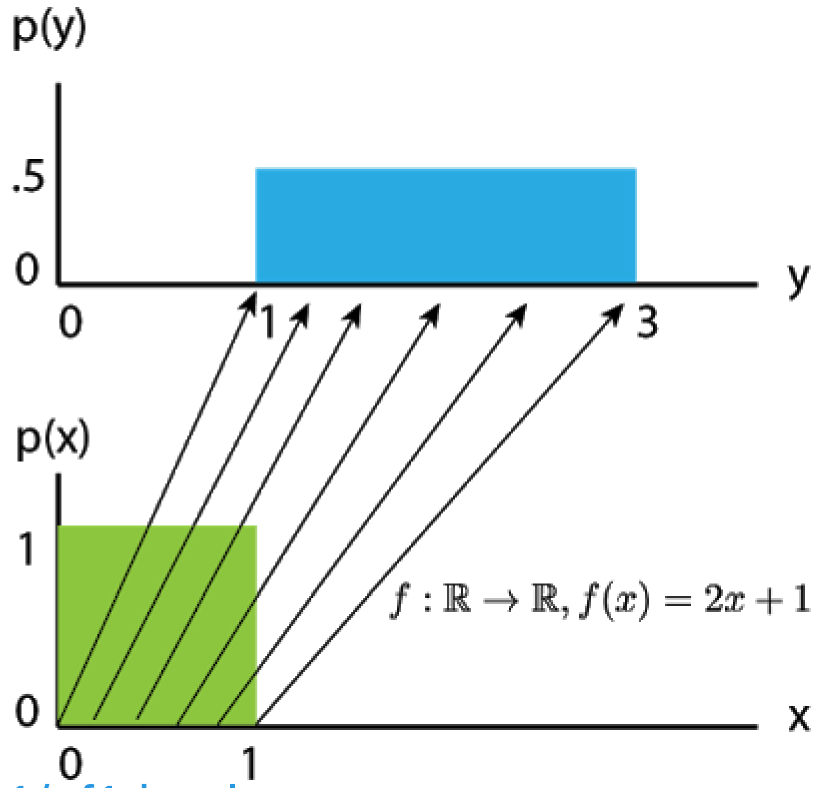
RealNVP contributions

- Coupling layers
- Masked convolutions
- Multi-scale architecture

Again: Change of variable formula

$$p_i(z_i) = p_{i-1}(f^{-1}(z_i)) \left| \det \frac{df_i}{dz_{i-1}} \right|^{-1}$$

- It must be easy to compute the inverse f^{-1}
- It must be easy to compute the determinant of the Jacobian $\det \frac{df_i}{dz_{i-1}}$



https://blog.evjang.com/2018/01/nf1.html

RealNVP: Coupling layers

- How to make computing the determinant easy?
- Have triangular matrices

$$A = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \det A = \prod_i a_{ii}$$

- So, let's design a transformation function that yields triangular Jacobians

RealNVP: Coupling layers

- Assume a D-dimensional input $x = x_{1:D}$
- Then, we split $x = [x_{1:d}, x_{d+1:D}]$ to have the bijective transformation

$$y_{1:d} = x_{1:d}$$
$$y_{d+1:D} = x_{d+1:D} \odot \exp\big(s(x_{1:d})\big) + t(x_{1:d})$$

- Basically, the first d dimensions remain unchanged, while the rest are modulated by the first d dimensions
- Let's check the Jacobian

$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \mathrm{diag}(\exp[s(x_{1:d})]) \end{bmatrix}$$

RealNVP: Coupling layers

$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \mathrm{diag}(\exp[s(x_{1:d})]) \end{bmatrix}$$

- What is the log-determinant?

RealNVP: Coupling layers

$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \mathrm{diag}(\exp[s(x_{1:d})]) \end{bmatrix}$$

- What is the log-determinant?

$$\log\det\frac{\partial y}{\partial x^T} = \sum_i \left(s(x_{1:d})\right)_j$$

- What convenient observation do we make?

RealNVP: Coupling layers

$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \text{diag}(\exp[s(x_{1:d})]) \end{bmatrix}$$

- What is the log-determinant?

$$\log\det\frac{\partial y}{\partial x^T} = \sum_i \left(s(x_{1:d})\right)_j$$

- What convenient observation do we make?
- Computing the log-determinant does not require compute the inverse or the determinant (or any other complex operation) of $s(x_{1:d})$
- So, $s(x_{1:d})$ can be as complex as we want
- For example?

RealNVP: Coupling layers

$$\frac{\partial y}{\partial x^T} = \begin{bmatrix} \mathbb{I}_d & 0 \\ \frac{\partial y_{d+1:D}}{\partial x_{1:d}^T} & \text{diag}(\exp[s(x_{1:d})]) \end{bmatrix}$$

- What is the log-determinant?

$$\log \det \frac{\partial y}{\partial x^T} = \sum_i \left(s(x_{1:d})\right)_j$$

- What convenient observation do we make?
- Computing the log-determinant does not require compute the inverse or the determinant (or any other complex operation) of $s(x_{1:d})$
- So, $s(x_{1:d})$ can be as complex as we want
- For example? Neural Networks

RealNVP: Coupling layers invertibility

- The inverse function is sort of trivial

$$y_{1:d} = x_{1:d}$$
$$y_{d+1:D} = x_{d+1:D} \odot \exp\big(s(d_{1:d})\big) + t(x_{1:d})$$
$$\Leftrightarrow$$
$$x_{1:d} = y_{1:d}$$
$$x_{d+1:D} = (y_{d+1:D} - t(x_{1:d})) \odot \exp\big(-s(d_{1:d})\big)$$

- Again, no complex operations required → neural networks ok
- Devil in the details:

RealNVP: Coupling layers invertibility

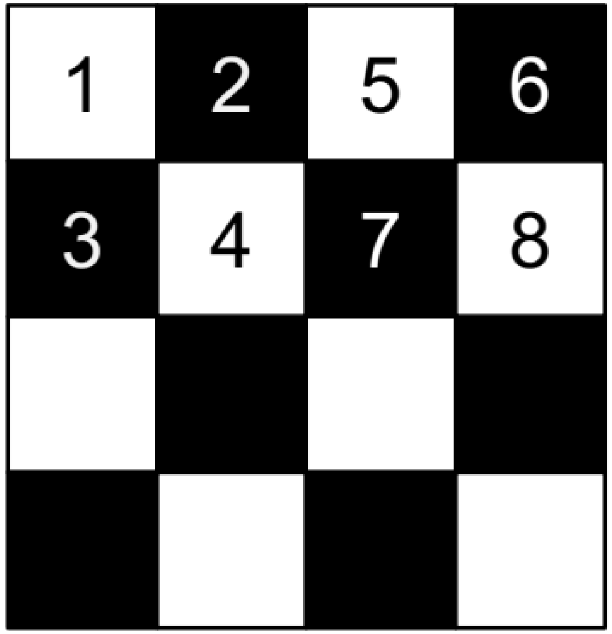
- The inverse function is sort of trivial

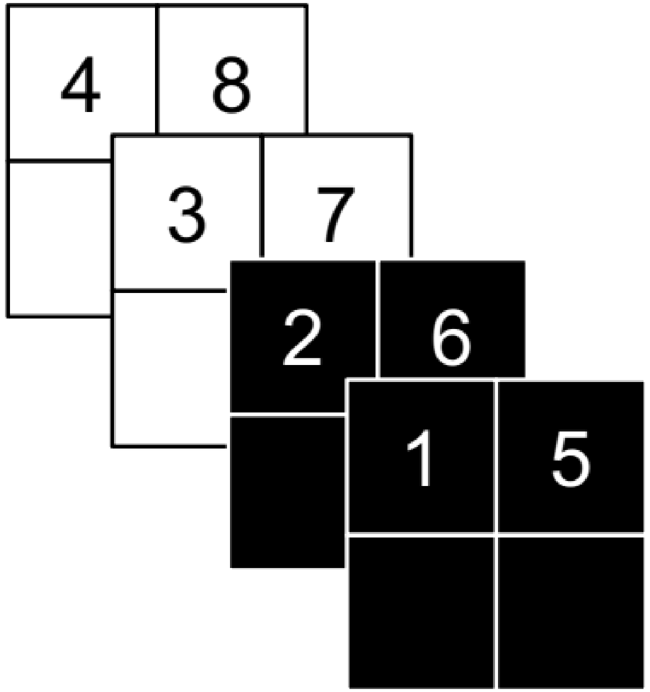
$$
\begin{gathered}
y_{1:d} = x_{1:d} \\
y_{d+1:D} = x_{d+1:D} \odot \exp\big(s(d_{1:d})\big) + t(x_{1:d}) \\
\Leftrightarrow \\
x_{1:d} = y_{1:d} \\
x_{d+1:D} = (y_{d+1:D} - t(x_{1:d})) \odot \exp\big(-s(d_{1:d})\big)
\end{gathered}
$$

- Again, no complex operations required → neural networks ok
- Devil in the details: The transformation must retain dimensionality
- So, no bottleneck → very large feature maps $t(x_{1:d})$

How to split? Masked convolutions

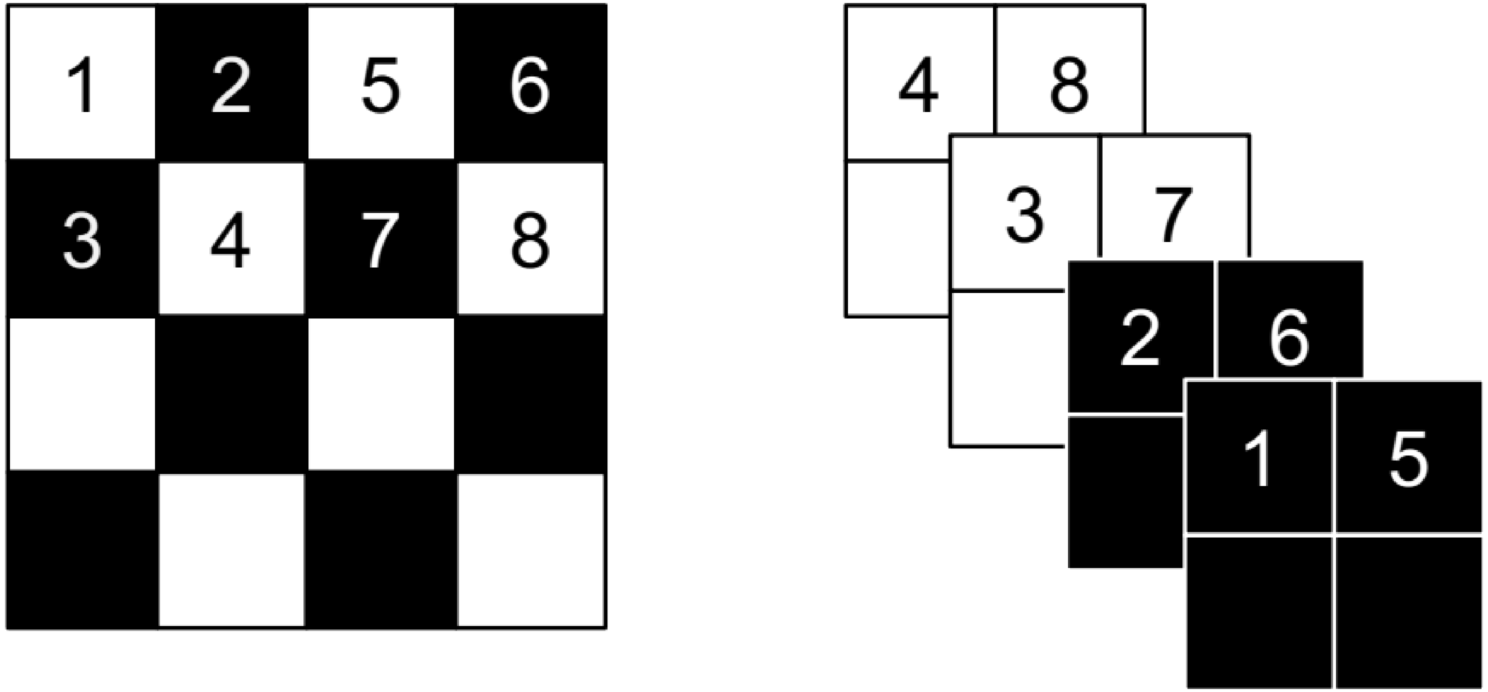
- Spatially: checkers pattern
- Channel-wise: half and half masking





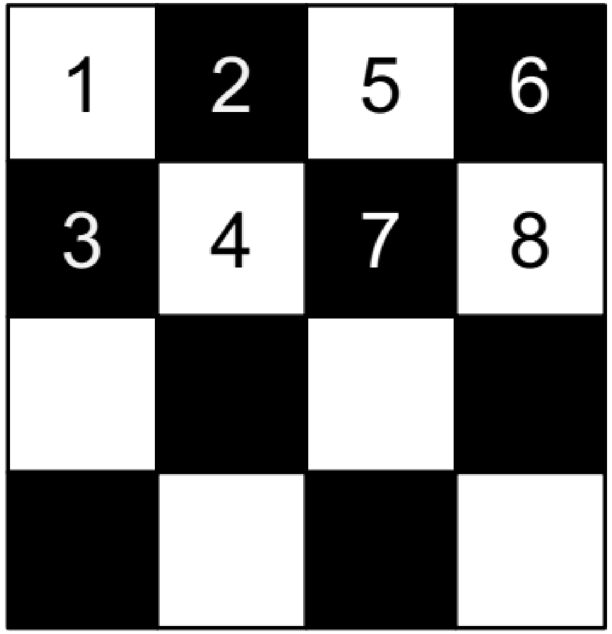
How to split? Masked convolutions

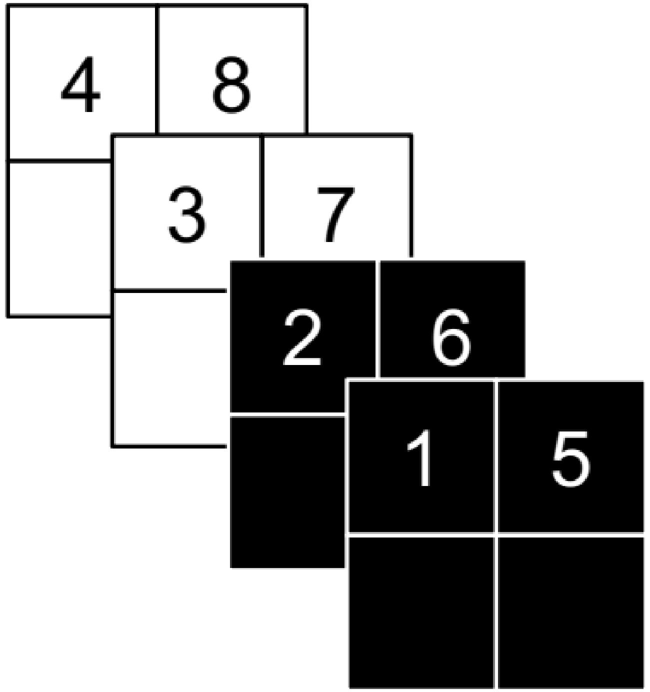
- Potential problem?



How to split? Masked convolutions

- Potential problem?
- Always the same dimensions in $y_{1:d} = x_{1:d}$. An easy fix?





RealNVP: reversing dimensions between coupling layers

- Do not use the same dimensions for the transformations when moving from one layer to the other

- Instead, alternate

- The Jacobian still remains tractable
- And the inverse also

Some results

Dataset	PixelRNN [46]	Real NVP	Conv DRAW [22]	IAF-VAE [34]
CIFAR-10	3.00	3.49	< 3.59	< 3.28
Imagenet (32×32)	3.86 (3.83)	4.28 (4.26)	< 4.40 (4.35)	
Imagenet (64×64)	3.63 (3.57)	3.98 (3.75)	< 4.10 (4.04)	
LSUN (bedroom)		2.72 (2.70)		
LSUN (tower)		2.81 (2.78)		
LSUN (church outdoor)		3.08 (2.94)		
CelebA		3.02 (2.97)		

Table 1: Bits/dim results for CIFAR-10, Imagenet, LSUN datasets and CelebA. Test results for CIFAR-10 and validation results for Imagenet, LSUN and CelebA (with training results in parenthesis for reference).

Qualitative results

Real samples

Generated samples

GLOW

Table 1: The three main components of our proposed flow, their reverses, and their log-determinants. Here, $\mathbf{x}$ signifies the input of the layer, and $\mathbf{y}$ signifies its output. Both $\mathbf{x}$ and $\mathbf{y}$ are tensors of shape $[h \times w \times c]$ with spatial dimensions (h, w) and channel dimension c. With (i, j) we denote spatial indices into tensors $\mathbf{x}$ and $\mathbf{y}$. The function $\mathtt{NN}()$ is a nonlinear mapping, such as a (shallow) convolutional neural network like in ResNets (He et al., 2016) and RealNVP (Dinh et al., 2016).

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i,j : \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$
Invertible 1×1 convolution. $\mathbf{W} : [c \times c]$. See Section 3.2.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$	$\forall i,j : \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log \lvert\det(\mathbf{W})\rvert$ or $h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$ (see eq. (10))
Affine coupling layer. See Section 3.3 and (Dinh et al., 2014)	$\mathbf{x}_a, \mathbf{x}_b = \mathtt{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \mathtt{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \mathtt{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \mathtt{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\mathtt{sum}(\log(\lvert\mathbf{s}\rvert))$

https://arxiv.org/abs/1807.03039

GLOW

Table 1: The three main components of our proposed flow, their reverses, and their log-determinants. Here, $\mathbf{x}$ signifies the input of the layer, and $\mathbf{y}$ signifies its output. Both $\mathbf{x}$ and $\mathbf{y}$ are tensors of shape $[h \times w \times c]$ with spatial dimensions (h, w) and channel dimension c. With (i, j) we denote spatial indices into tensors $\mathbf{x}$ and $\mathbf{y}$. The function $\mathtt{NN}()$ is a nonlinear mapping, such as a (shallow) convolutional neural network like in ResNets (He et al., 2016) and RealNVP (Dinh et al., 2016).

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1	Replaces BatchNorm that needs large minibatches		
Invertible 1×1 convolution. $\mathbf{W} : [c \times c]$. See Section 3.2	$\forall i, j : \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$ Generalization of permutation operation in RealNVP	$\forall i, j : \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log\lvert\det(\mathbf{W})\rvert$ (see eq. (10))
Affine coupling layer. See Section 3.3 and (Dinh et al., 2014)	$\mathbf{x}_a, \mathbf{x}_b = \mathtt{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \mathtt{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \mathtt{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \mathtt{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\mathtt{sum}(\log(\lvert\mathbf{s}\rvert))$

https://arxiv.org/abs/1807.03039

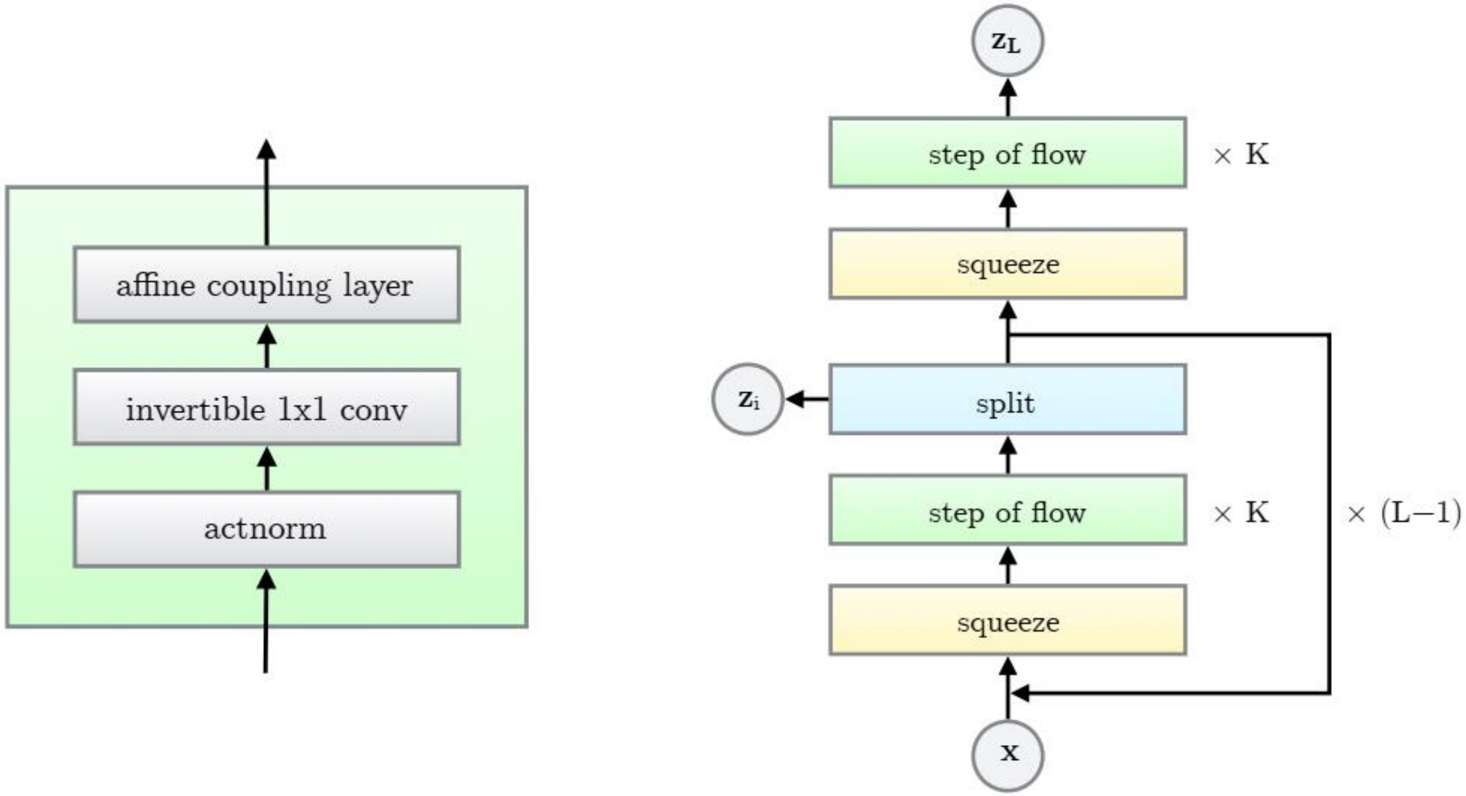
GLOW

Table 1: The three main components of our proposed flow, their reverses, and their log-determinants. Here, $\mathbf{x}$ signifies the input of the layer, and $\mathbf{y}$ signifies its output. Both $\mathbf{x}$ and $\mathbf{y}$ are tensors of shape $[h \times w \times c]$ with spatial dimensions (h, w) and channel dimension c. With (i, j) we denote spatial indices into tensors $\mathbf{x}$ and $\mathbf{y}$. The function $\mathtt{NN}()$ is a nonlinear mapping, such as a (shallow) convolutional neural network like in ResNets (He et al., 2016) and RealNVP (Dinh et al., 2016).

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i,j : \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$
Invertible 1×1 convolution. $\mathbf{W} : [c \times c]$. See Section 3.2.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$	$\forall i,j : \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log \lvert\det(\mathbf{W})\rvert$ or $h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$ (see eq. (10))
Affine coupling layer. See Section 3.3 and (Dinh et al., 2014)	$\mathbf{x}_a, \mathbf{x}_b = \mathtt{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \mathtt{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \mathtt{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \mathtt{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\mathtt{sum}(\log(\lvert\mathbf{s}\rvert))$

https://arxiv.org/abs/1807.03039

GLOW pipeline



(a) One step of our flow.

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

Figure 2: We propose a generative flow where each step (left) consists of an *actnorm* step, followed by an invertible 1×1 convolution, followed by an affine transformation (Dinh et al., 2014). This flow is combined with a multi-scale architecture (right). See Section 3 and Table 1.

Some results

Table 2: Best results in bits per dimension of our model compared to RealNVP.

Model	CIFAR-10	ImageNet 32x32	ImageNet 64x64	LSUN (bedroom)	LSUN (tower)	LSUN (church outdoor)
RealNVP	3.49	4.28	3.98	2.72	2.81	3.08
Glow	**3.35**	**4.09**	**3.81**	**2.38**	**2.46**	**2.67**

Some visual examples

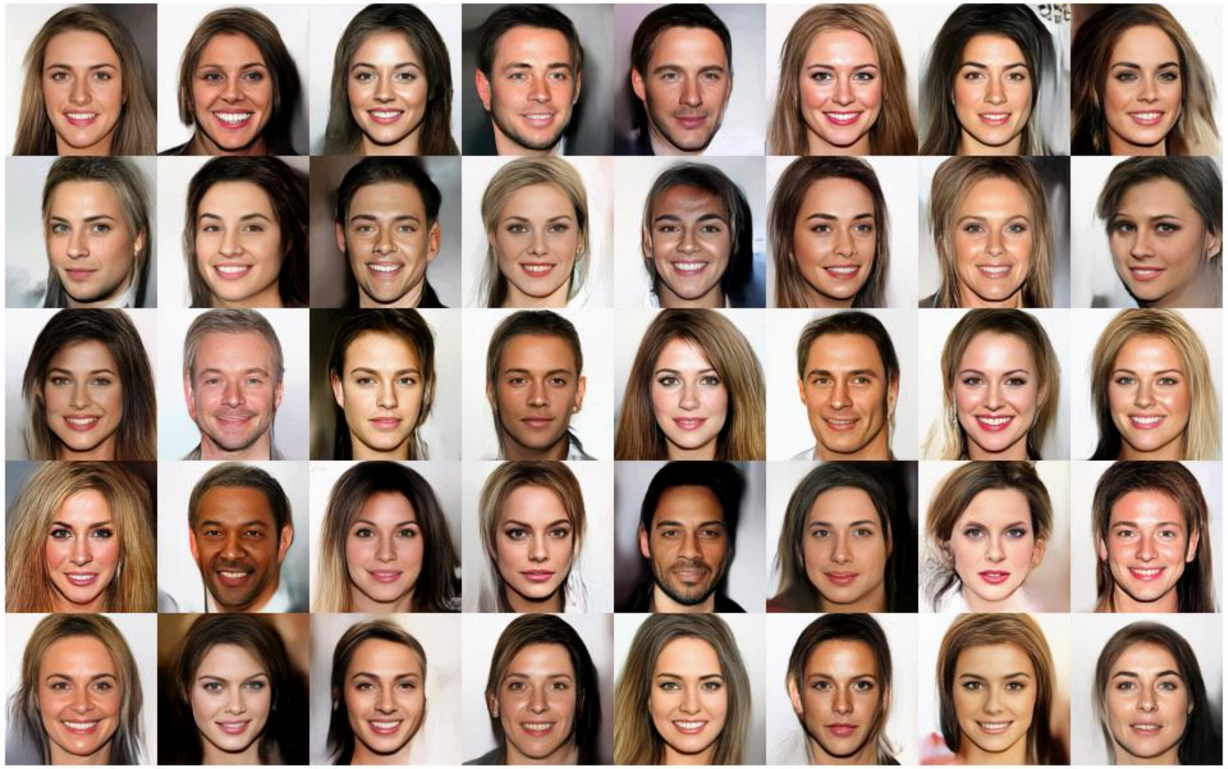


Figure 4: Random samples from the model, with temperature 0.7

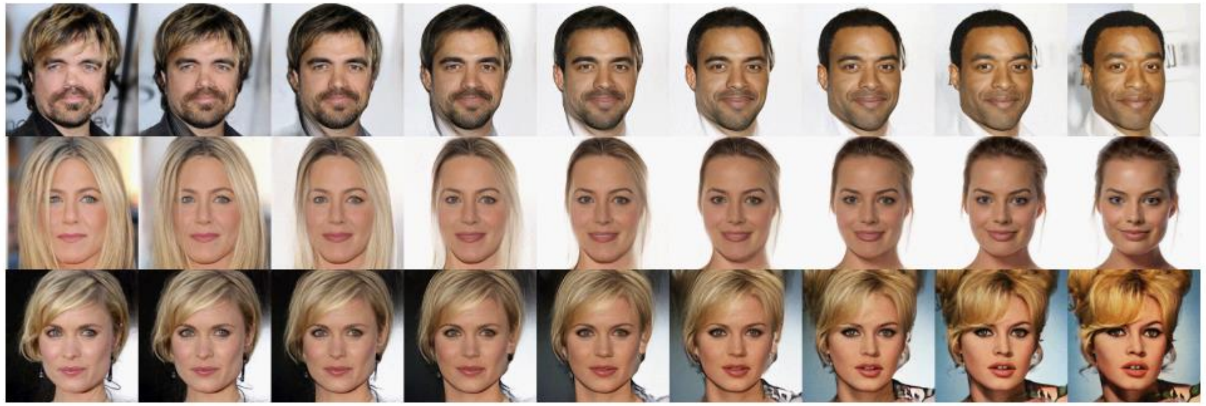
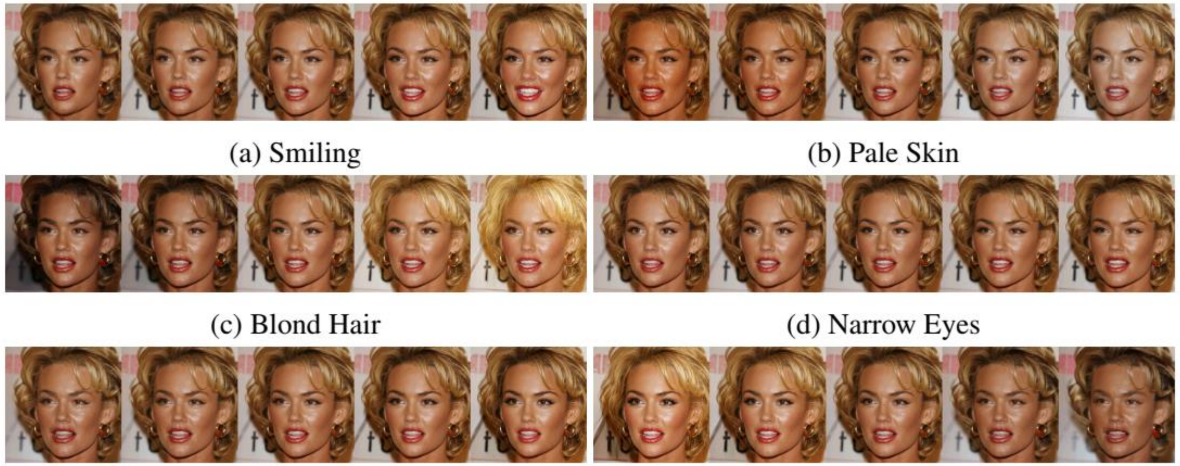


Figure 5: Linear interpolation in latent space between real images



(a) Smiling

(b) Pale Skin

(c) Blond Hair

(d) Narrow Eyes

(e) Young

(f) Male

FLOW++

- In reality, images are continuous signals but recorded digitally (integers)
- Fitting a continuous density function will therefore encourage a degenerate solution that puts all probability mass on discrete data points

- An easy solution is "dequantization" (Uria et al. 2013, Dinh et al., 2016, Salimans et al., 2017)
- If your image is x you simply add a bit of noise, so that x does not gather around specific (discrete) values

$$y = x + u, u \sim \text{Uniform}(0, 1)$$

https://arxiv.org/abs/1902.00275

FLOW++ dequantization

- It can be shown that dequantization optimizes a lower bound on the original discrete data

$$P_{model}(x) = \int_{[0,1)^D} p_{model}(x+u)du$$

- So, since we are anyways using a (uniform) distribution u, why not … ?

FLOW++ dequantization

- It can be shown that dequantization optimizes a lower bound on the original discrete data

$$P_{model}(x) = \int_{[0,1)^D} p_{model}(x+u)du$$

- So, since we are anyways using a (uniform) distribution u, why not … ?

- Learn the optimal noise distribution. How?

FLOW++ dequantization

- It can be shown that dequantization optimizes a lower bound on the original discrete data

$$P_{model}(x) = \int_{[0,1)^D} p_{model}(x+u)du$$

- So, since we are anyways using a (uniform) distribution u, why not … ?

- Learn the optimal noise distribution. How?
- Variational Inference to the rescue → Variational Dequantization

Variational Dequantization

$$\mathbb{E}_{x\sim P_{data}}[\log P_{model}(x)] \geq \mathbb{E}_{x\sim P_{data}, \varepsilon\sim p(\varepsilon)}\left[\log \frac{p_{model}\left(x + q_x(\varepsilon)\right)}{\pi(\varepsilon)\left|{}^{\partial q_x}/_{\partial \varepsilon}\right|^{-1}}\right]$$

- The noise model $q_x(\varepsilon)$ is implemented as a flow-based generative model
- As p_{model} is also flow-based, computing the Jacobians is possible

Some results

Table 1. Unconditional image modeling results in bits/dim

Model family	Model	CIFAR10	ImageNet 32x32	ImageNet 64x64
Non-autoregressive	RealNVP (Dinh et al., 2016)	3.49	4.28	–
	Glow (Kingma & Dhariwal, 2018)	3.35	4.09	3.81
	IAF-VAE (Kingma et al., 2016)	3.11	–	–
	Flow++ (ours)	**3.08**	**3.86**	**3.69**
Autoregressive	Multiscale PixelCNN (Reed et al., 2017)	–	3.95	3.70
	PixelCNN (van den Oord et al., 2016b)	3.14	–	–
	PixelRNN (van den Oord et al., 2016b)	3.00	3.86	3.63
	Gated PixelCNN (van den Oord et al., 2016c)	3.03	3.83	3.57
	PixelCNN++ (Salimans et al., 2017)	2.92	–	–
	Image Transformer (Parmar et al., 2018)	2.90	3.77	–
	PixelSNAIL (Chen et al., 2017)	2.85	3.80	3.52

Some visual examples

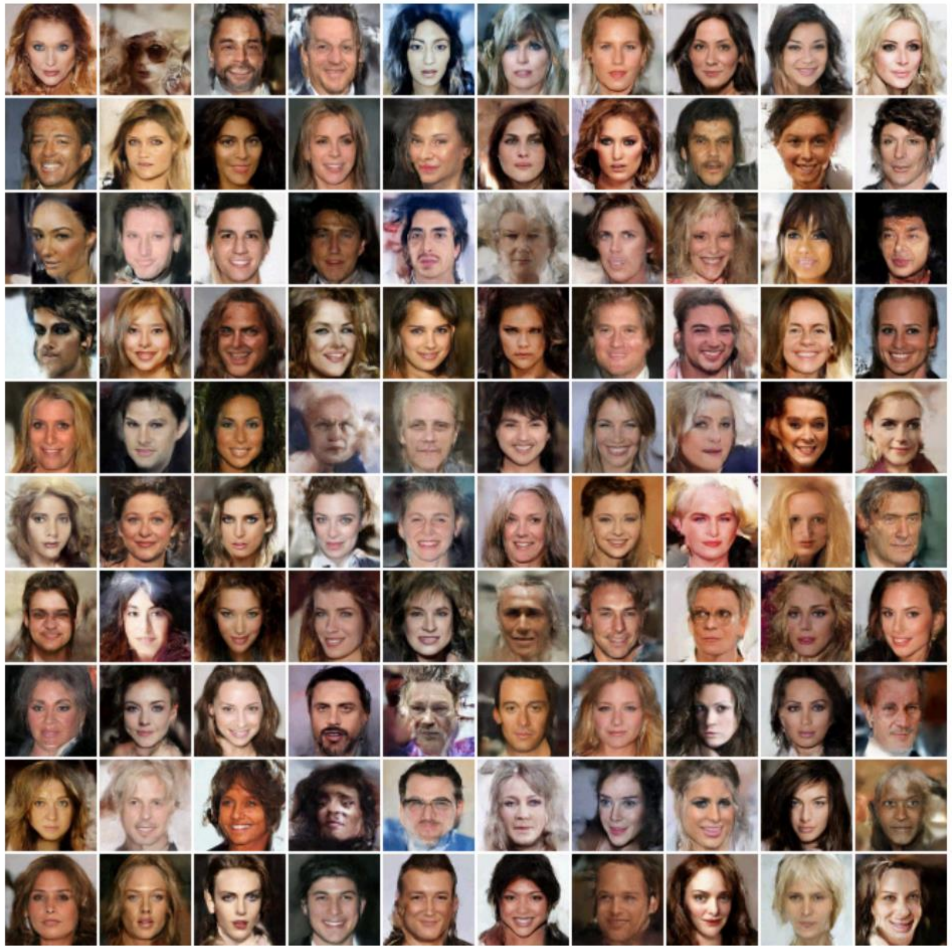
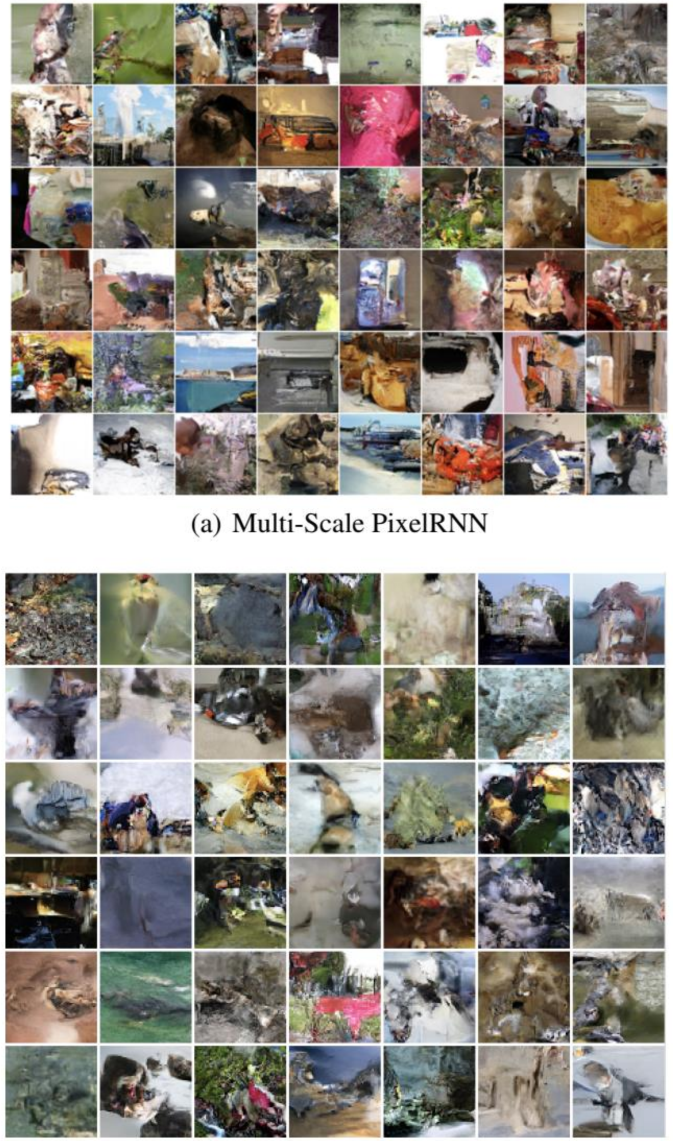


Figure 4. Samples from Flow++ trained on 5-bit 64x64 CelebA, without low-temperature sampling.



(a) Multi-Scale PixelRNN

(b) Flow++

Figure 5. 64x64 ImageNet Samples. Top: samples from Multi-Scale PixelRNN (van den Oord et al., 2016b). Bottom: samples from Flow++. The diversity of samples from Flow++ matches the diversity of samples from PixelRNN with multi-scale ordering.

A recap

- Three families of likelihood-based generative models
  - Variational autoencoders
  - Autoregressive models
  - Flow-based models

- Autoregressive and flow-based models are exact-likelihood models
  - They compute $p(x)$ directly, not an approximation or an estimation or a bound

A summary of properties

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

17

J. Tomczak's lecture from April, 2019

Summary

- Exact likelihood models
  - Autoregressive Models
  - Non-autoregressive flow-based models
- Autoregressive Models
  - NADE, MADE, PixelCNN, PixelCNN++, PixelRNN
- Normalizing Flows
- Non-autoregressive flow-based models
  - RealNVP
  - Glow
  - Flow++