

10707

Deep Learning: Spring 2021

Andrej Risteski

Machine Learning Department

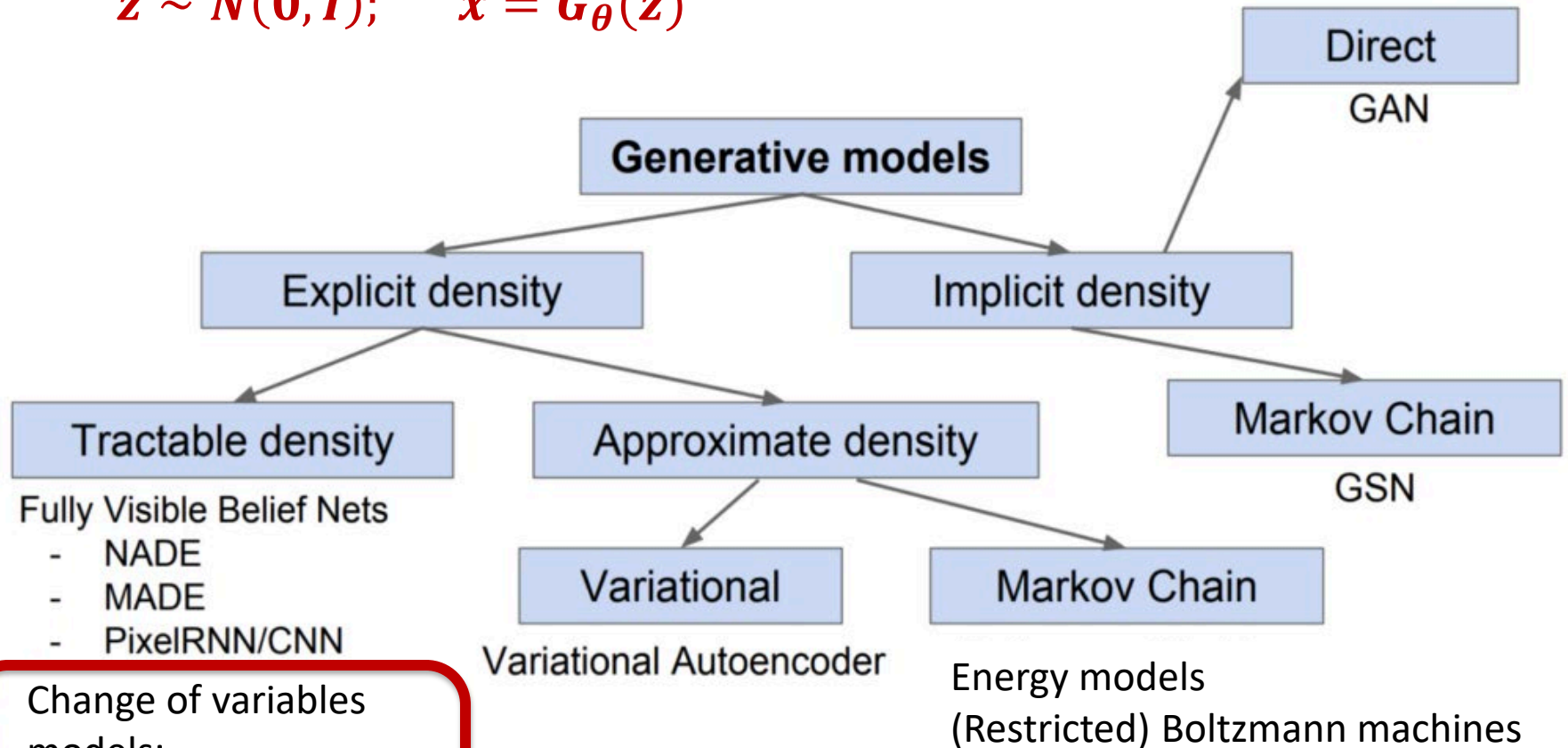
Lecture 17:

Normalizing flows and autoregressive
models

Part I: Normalizing flows

They model the distribution for x as an invertible transformation of an (easy) distribution over z :

$$z \sim N(0, I); \quad x = G_{\theta}(z)$$



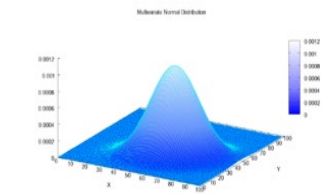
Change of variables models:

- (Nonlinear) ICA
- Normalizing flows

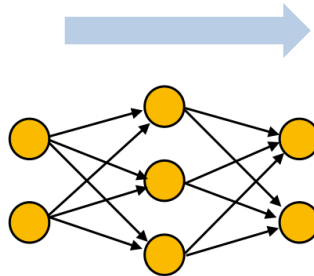
The GAN paradigm (Goodfellow et al. '14)

Goal: **Learn** a distribution close to some distribution we have few samples from. (Additionally, we will be able to sample efficiently from distribution.)

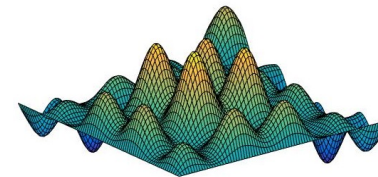
Approach: **Fit** distribution P_g parametrized by **neural network g**



$$Z \sim N(0, I_{k \times k})$$



Neural network $g(\cdot)$



$$X = g(Z)$$

The pros and cons of GANs

Pros

Photorealism: photorealistic images, even w/ **relatively small models**.

Efficient sampling: easy to draw samples from model (unlike e.g. energy models).

Cons

Unstable training: min-max problem – typically optimization much less stable than pure minimization.

Mode collapse: training only recovers some of the “modes” of the underlying distribution.

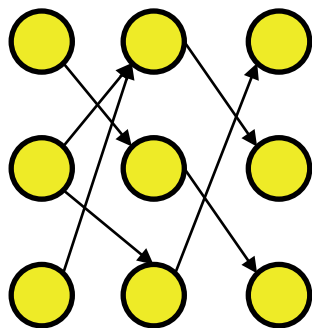
Evaluation: no likelihood, so hard to evaluate fit.



Middle ground: “Invertible GANs/Normalizing Flows”

Can we “marry” likelihood models w/ GANs?

Suppose generator $g: \mathbb{R}^d \rightarrow \mathbb{R}^d$ were **invertible**.



Recall from the prev. lecture: if we denote by $\phi(z)$ the density of z under the standard Gaussian, by the change of variables formula:

$$P_g(x) = \phi(g^{-1}(x)) |\det(J_x(g^{-1}(x)))|$$

Hence, we can write down the likelihood in terms of the parameters of g^{-1} under this model!



Middle ground:

“Invertible GANs/Normalizing Flows”

$$P_g(x) = \phi(g^{-1}(x)) | \det(J_x(g^{-1}(x))) |$$

Hence, denoting $g^{-1} = f_\theta$, for some family of parametric functions $\{f_\theta, \theta \in \Theta\}$, the max-likelihood estimator solves

$$\max_{\theta} \sum_{i=1}^N \log \phi(f_\theta(x_i)) + \log | \det(J_x(f_\theta(x_i))) |$$


If we can evaluate and differentiate the above objective efficiently, we can do gradient-based likelihood fitting.



Choosing invertible transforms

Note that since the change-of-variables formula composes, so if $f_\theta = f_1 \circ f_2 \circ \dots \circ f_L$, we have

$$\log p_\theta(x) = \sum_{i=1}^N \log \phi(f_\theta(x_i)) + \sum_{k=1}^L \log |\det(J_x(f_k(h_k(x_i))))|$$

Value of k-th layer 

So, if we can design a “simple” family of invertible transforms, we can just keep composing it.

Try 1: *General linear maps.*

Poor representational power: composition of linear maps is linear. If $x = Az$, and z is sampled from a Gaussian – x is Gaussian too.

Inefficient: Evaluating determinant of a $d \times d$ matrix takes $O(d^3)$ time – infeasible.

Choosing invertible transforms

Try 2: *Elementwise (possibly non-linear) maps.*

Suppose that $f_\theta(x) = (f_\theta(x_1), f_\theta(x_2), \dots, f_\theta(x_d))$

Efficient evaluation: Determinant is diagonal (since $\frac{\partial f_\theta(x_i)}{\partial x_j} = 0$, for $i \neq j$), so

$$\det(J_x(f_\theta(x))) = \prod_i \frac{\partial f_\theta(x_i)}{\partial x_i}.$$

Poor representational power: Transforms don't “combine” coordinates.

But, even if a matrix is triangular, Jacobian is just the product of the diagonals!!

Choosing invertible transforms

Try 3: *NICE/RealNVP (Non-linear Independent Component Estimation)*

Divide the coordinates of x into two sub-vectors with half the coords: $x_{1:\frac{d}{2}}, x_{\frac{d}{2}+1,d}$

Divide the coordinates of $z := f_\theta(x)$ into two sub-vectors, $z_{1:\frac{d}{2}}, z_{\frac{d}{2}+1,d}$ and set:

$$z_{1:\frac{d}{2}} = x_{1:\frac{d}{2}}$$

$$z_{\frac{d}{2}+1,d} = x_{\frac{d}{2}+1,d} \odot \exp\left(s_\theta(x_{1:d/2})\right) + t_\theta(x_{1:d/2})$$

When is this invertible, and is the Jacobian efficiently calculated?

Choosing invertible transforms

Try 3: *NICE/RealNVP (Non-linear Independent Component Estimation)*

$$z_{1:d/2} = x_{1:d/2}$$

$$z_{\frac{d}{2}+1,d} = x_{\frac{d}{2}+1,d} \odot \exp\left(s_{\theta}(x_{1:d/2})\right) + t_{\theta}(x_{1:d/2})$$

$$J_x(f_{\theta}(x)) = \begin{bmatrix} I & 0 \\ \frac{\partial \mathbf{z}_{d/2:d}}{\partial \mathbf{x}_{1:d/2}} & \text{diag}\left(\exp(\mathbf{s}_{\theta}(\mathbf{x}_{1:d/2}))\right) \end{bmatrix}$$

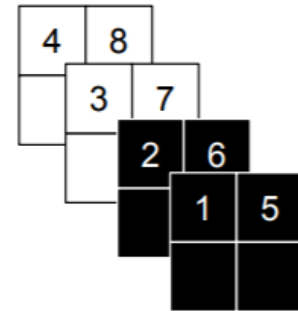
The determinant of a triangular matrix is the product of the diagonals!

$$\text{Hence, } \det J_x(f_{\theta}(x)) = \prod_i \exp\left(s_{\theta}(x_{1:d/2})\right)_i$$

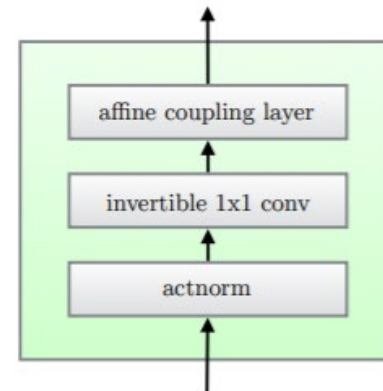
If t_{θ}, s_{θ} is say, a neural net, easy to evaluate and take derivatives.

How to choose partitions?

NICE (Dinh et al '14), RealNVP (Dinh et al '16): The choice of partitions is fixed, checkerboard of channel-wise.



Glow (Kingma et al '18): add trained linear transforms between affine coupling layers – i.e. a generalization of a “learned” permutation



Some samples

NICE/RealNVP (Non-linear Independent Component Estimation)

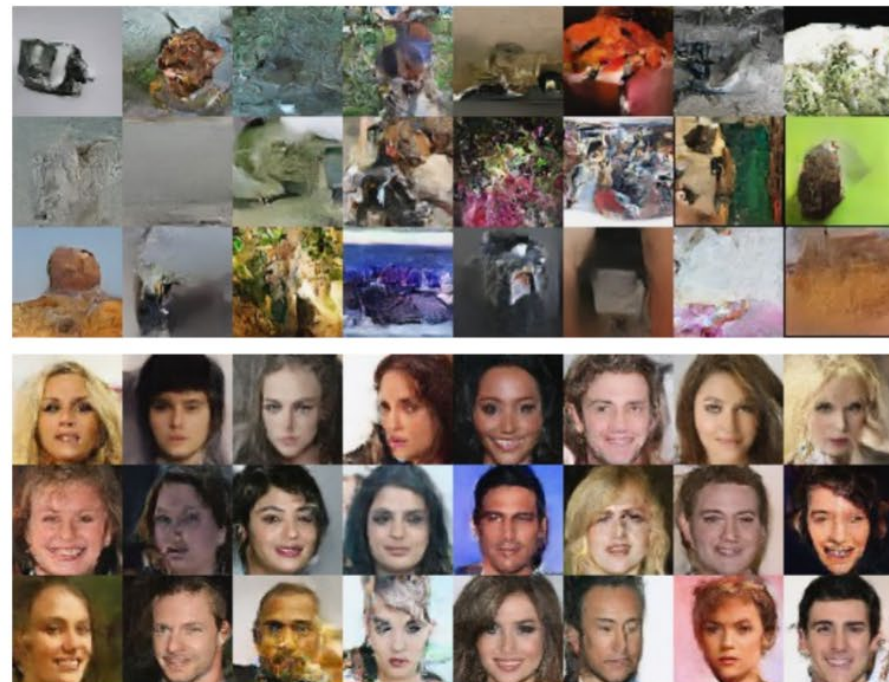
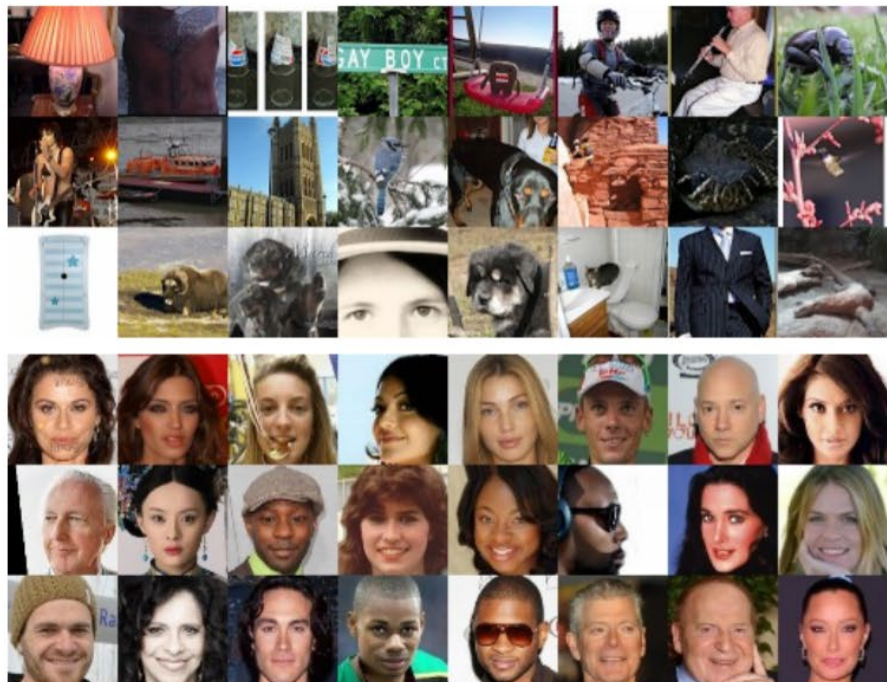


Figure from “*Density estimation using Real NVP*” by Dinh et al ‘16

Some samples

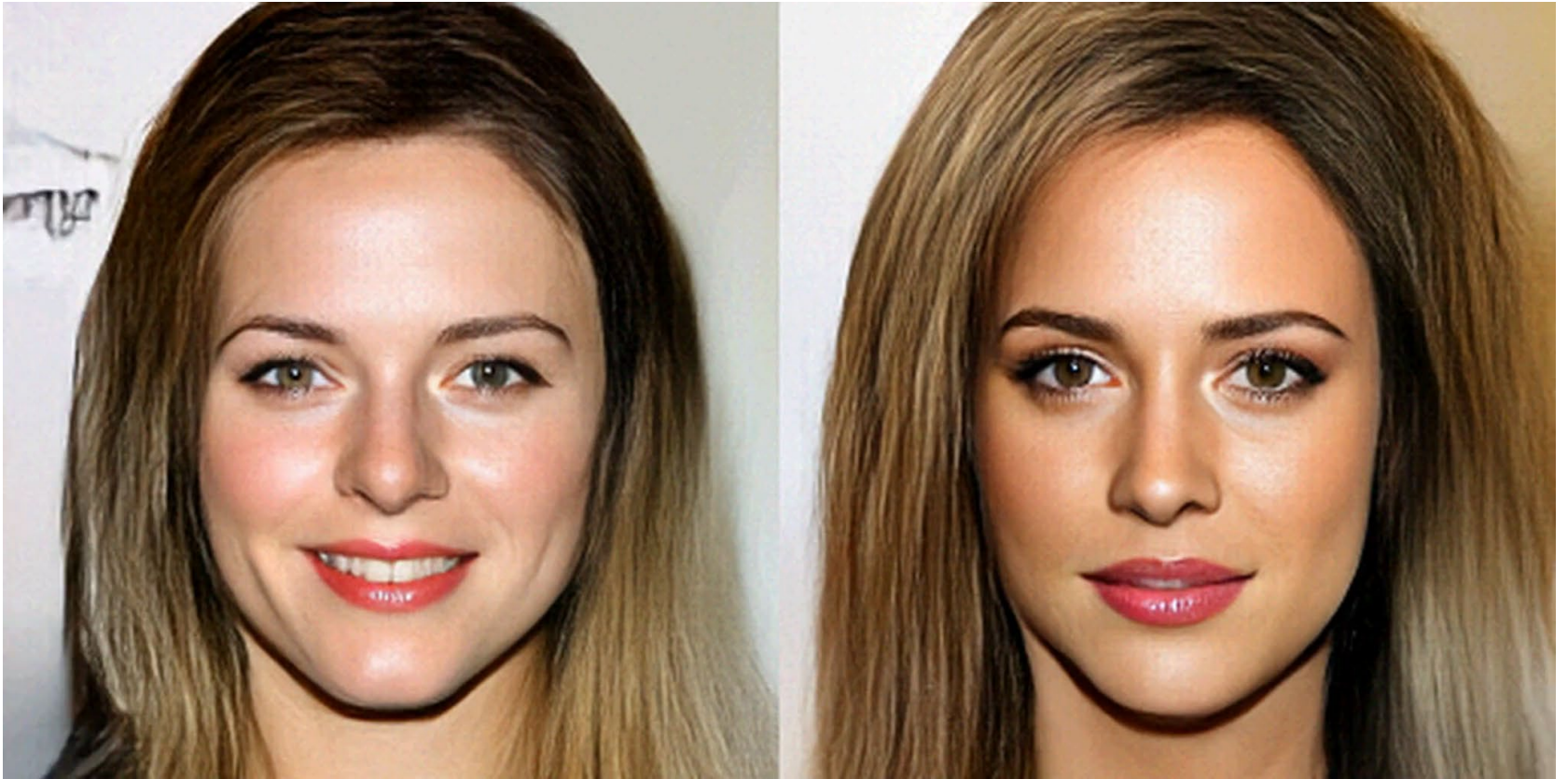
Glow



Figure 5: Linear interpolation in latent space between real images

Figure from (Kingma et al '18)

Some samples



<https://openai.com/blog/glow/>

The pros and cons of flow models

Pros

Photorealism: photorealistic images.

Efficient sampling: easy to draw samples from model.

Stabl(er) training: likelihood objective, pure minimization.

Evaluation: easier, comes with likelihood.

Cons

Extremely large: in practice, good models need to be *extremely* large (Glow: 40 GPUs for ~week)

Model depth: training gets harder as models are typically very deep. (Glow: ~1200 layers in total)

Ill-conditioned: Jacobian is close-to-singular, so objective blows up.



Some representational reasons why

(Koehler-Mehta-Risteski '20), half-baked conclusions:

Universal approximation with ill-conditioned affine couplings: any sufficiently nice distribution can be approximated arbitrarily closely with an (*ill-conditioned*) affine coupling network.

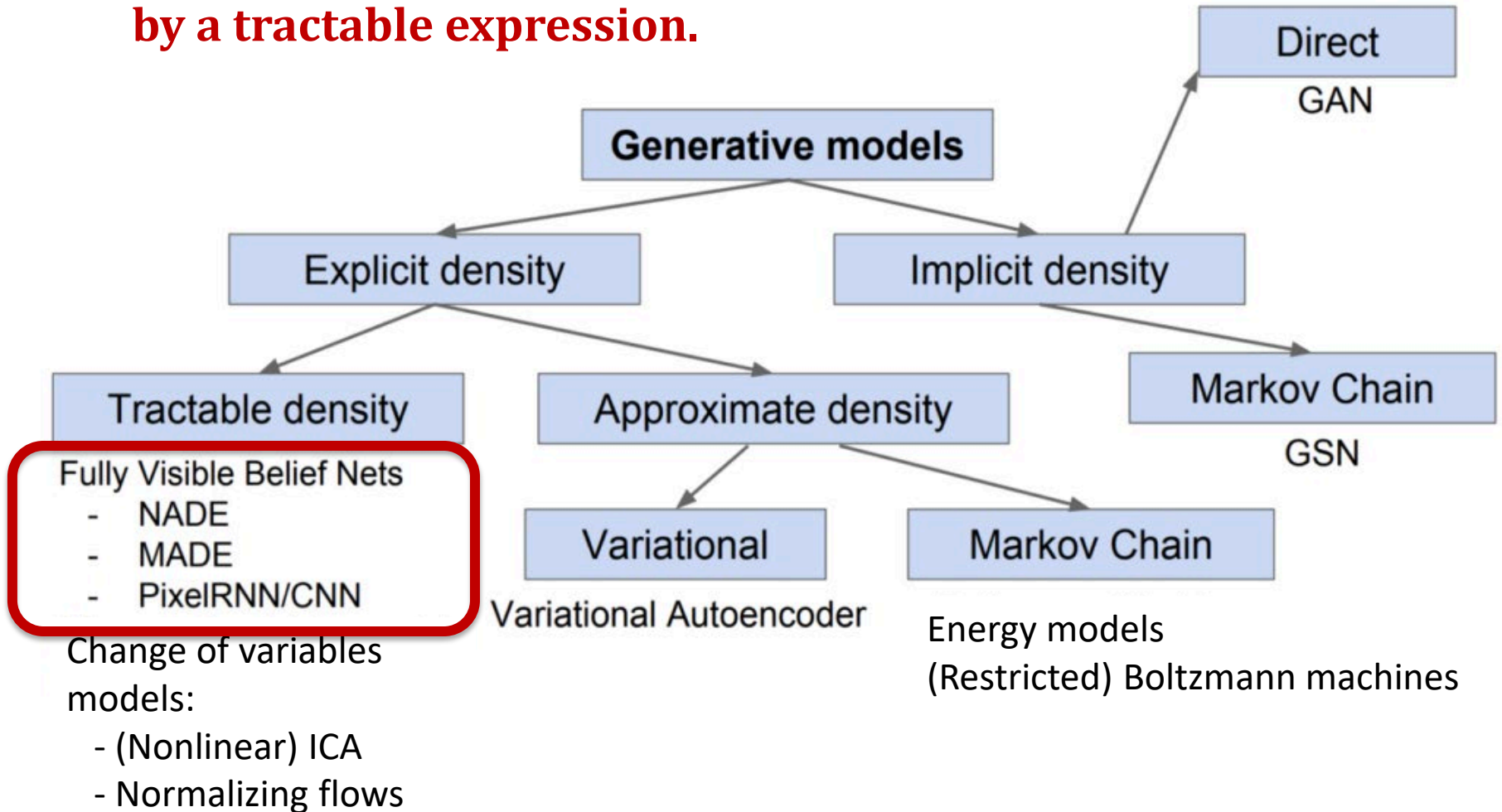
Learned permutations don't help that much: any linear transform can be simulated via 8 affine coupling layers with any fixed partition.

Inevitable increase in depth: there is a distribution representable as a **small and shallow** generator, s.t. a ***much deeper*** invertible model is necessary to approximate it (*architecture-independent*).



Part II: Autoregressive models

Autoregressive models, they model the distribution $p_{\theta}(x_t|x_{<t})$ by a tractable expression.



Sequential structure in data

Often times, data we are interested has “**sequential**” structure:

For instance:

☞ *Word* in sentence depends on surrounding words.

☞ *Sounds* in speech depend on surrounding sounds.

☞ *Pixel* in image depends on surrounding pixels.

It makes sense to factor joint distribution of data as

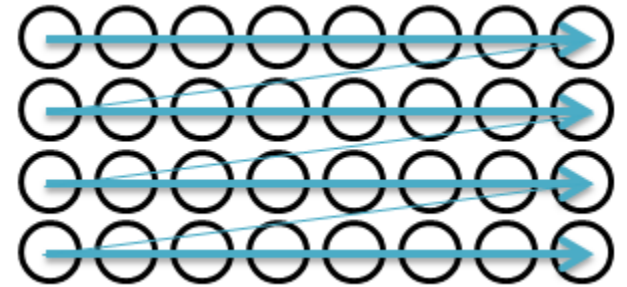
$$p(x_1, x_2, \dots, x_t) = p(x_i | x_{<i})$$

We will model $p(x_i | x_{<i})$ s.t. we can sample from/learn the model efficiently.

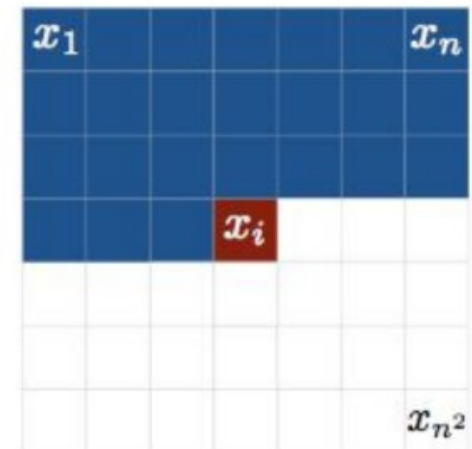
Sequential structure in data

Requires fixing an order. In text, this is the obvious one. Same in sound/speech.

In images, the typical ordering is the “raster ordering”



Thus, the notation $x_{<i}$ will denote:



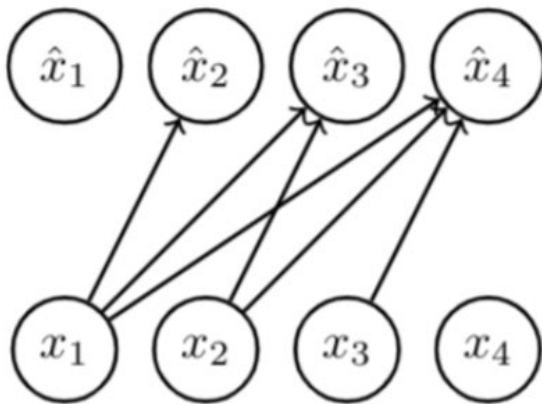
Fully visible sigmoid belief network

Let's assume data is binary (e.g. MNIST).

The “obvious” parametrization for $p(x_i|x_{<i})$:

Different parameters
for all i

$$p(x_i|x_{<i}) = \sigma \left(\sum_{j=1}^i \theta_j^i x_j \right)$$



Sampling: obvious ancestral sampling

$$\widehat{x}_1 \sim p(x_1)$$

$$\widehat{x}_i \sim p(x_i|\widehat{x}_{<i})$$

Training: log-likelihood/gradients are explicit.

Fully visible sigmoid belief network

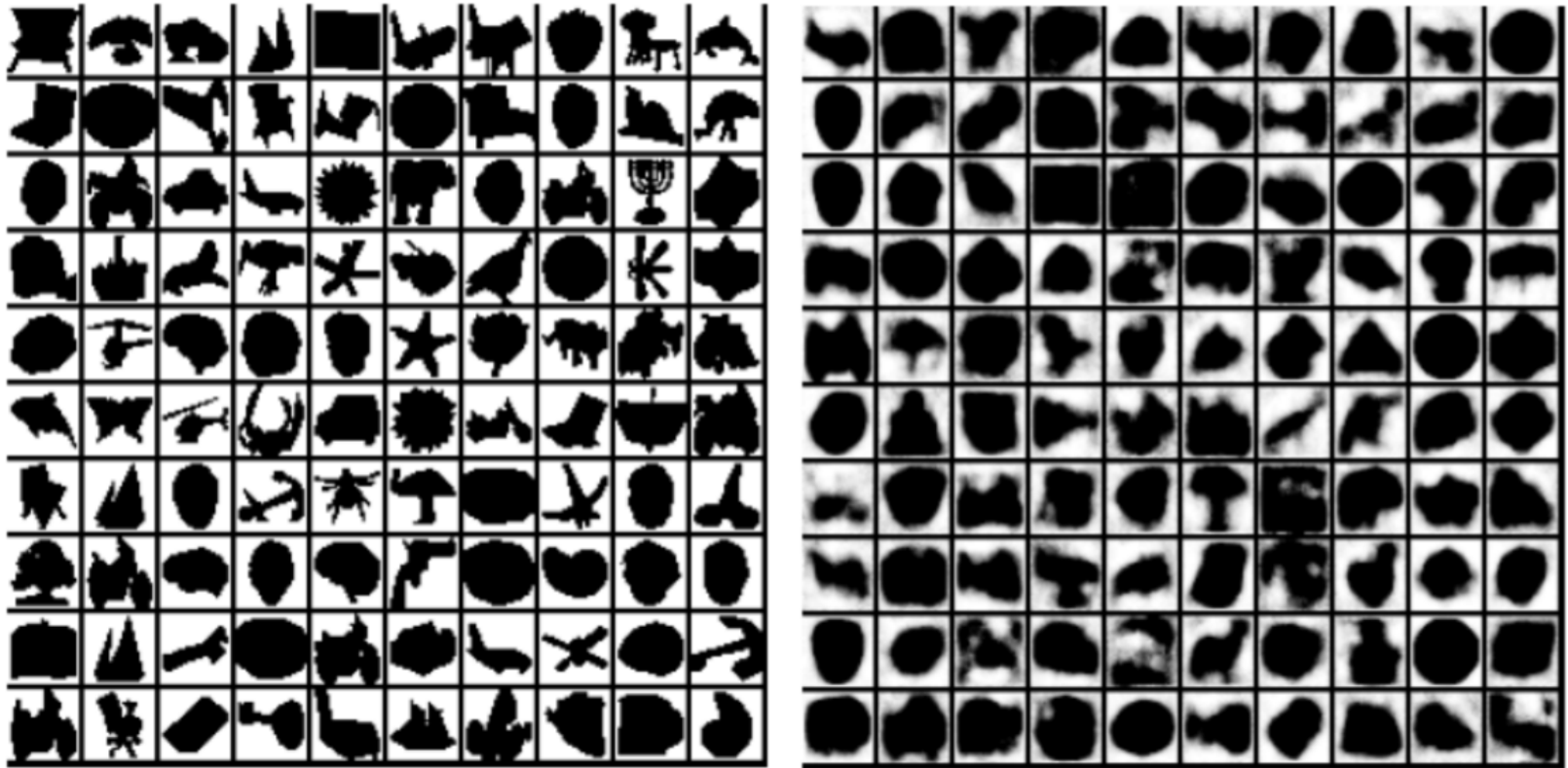


Figure from *Learning Deep Sigmoid Belief Networks with Data Augmentation*, Gan et al' 2015

NADE: Neural Autoregressive Density Estimation

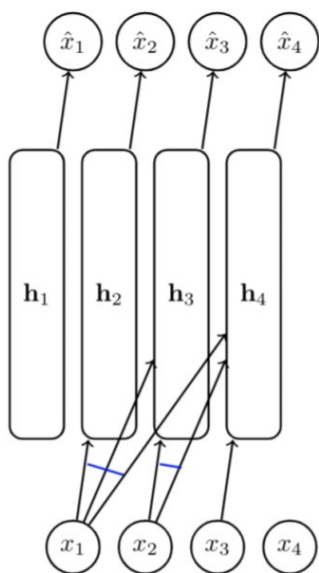
Let's assume data is binary (e.g. MNIST).

The “slightly more neural” parametrization for $p(x_i|x_{<i})$:

$$h_i = \sigma(A_i x_{<i} + c_i)$$

$$p(x_i|x_{<i}) = \sigma(\alpha_i^T h_i + b_i)$$

One-hidden layer net,
sigmoid activation σ



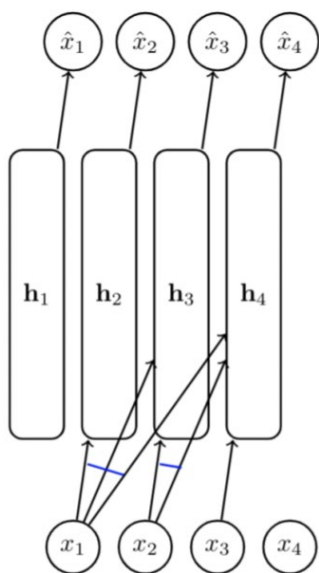
Sampling, training: same as before

NADE: Neural Autoregressive Density Estimation

Weight-tied variants:

$$h_i = \sigma(W_{\cdot, < i} x_{< i} + c)$$

$$p(x_i | x_{< i}) = \sigma(\alpha_i^T h_i + b_i)$$



e.g.

$$h_2 = \sigma \left(\underbrace{\begin{pmatrix} \vdots \\ \textcolor{blue}{w}_1 \\ \vdots \end{pmatrix}}_{A_2} x_1 \right) \quad h_3 = \sigma \left(\underbrace{\begin{pmatrix} \vdots \\ \textcolor{blue}{w}_1 & \textcolor{red}{w}_2 \\ \vdots \end{pmatrix}}_{A_3} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) \quad h_4 = \sigma \left(\underbrace{\begin{pmatrix} \vdots \\ \textcolor{blue}{w}_1 & \textcolor{red}{w}_2 & w_3 \\ \vdots \end{pmatrix}}_{A_3} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \right)$$

NADE: Neural Autoregressive Density Estimation



Figure from *Learning Deep Sigmoid Belief Networks with Data Augmentation*, Gan et al' 2015

MADE: Masked Autoencoder for Distribution Estimation

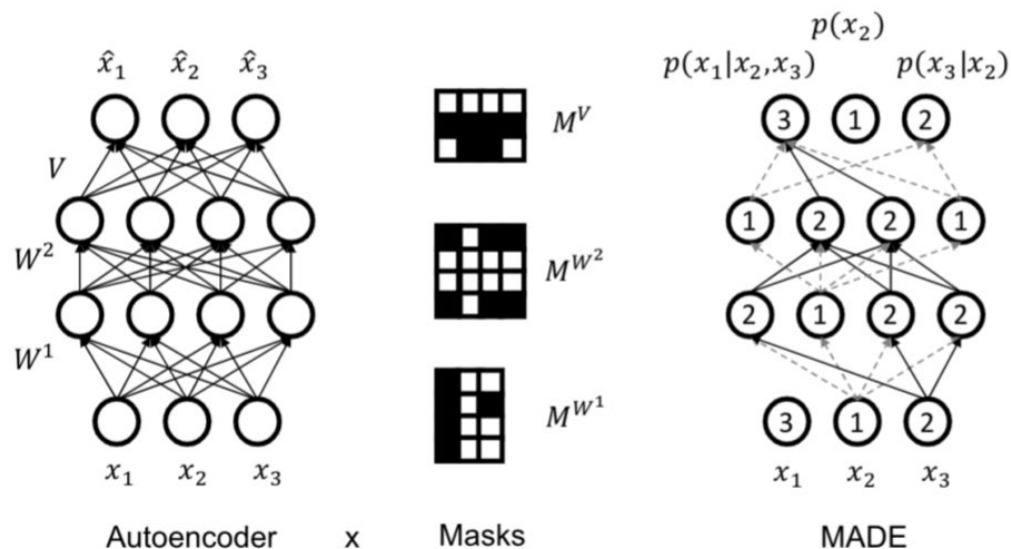
Strategy: turn an autoencoder into an autoregressive distribution modeler.

Recall, an autoencoder can be used to specify $p(\hat{x}|x)$ (via the reconstruction loss).

Suppose for *some* ordering of the input coordinates, \hat{x}_1 is independent of x , \hat{x}_2 only depends on x_1 , ... \hat{x}_t only on $x_{<t}$ -- we can sample autoregressively.

Solution: use masks to ensure this. Choose an ordering, and only allow weights that respect ordering.

MADE: Masked Autoencoder for Distribution Estimation



To zero out connections: mask via entrywise multiplication, e.g. for one layer:

$$\begin{aligned} \mathbf{h}(\mathbf{x}) &= \mathbf{g}(\mathbf{b} + (\mathbf{W} \odot \mathbf{M}^{\mathbf{W}})\mathbf{x}) \\ \hat{\mathbf{x}} &= \text{sigm}(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}(\mathbf{x})) \end{aligned}$$

To construct appropriate masks: for each node k , pick index $m(k)$ uniformly at random in $[1, D-1]$: denoting which inputs node depends on.

$$M_{k,d}^{\mathbf{W}} = 1_{m(k) \geq d} = \begin{cases} 1 & \text{if } m(k) \geq d \\ 0 & \text{otherwise,} \end{cases} \quad M_{d,k}^{\mathbf{V}} = 1_{d > m(k)} = \begin{cases} 1 & \text{if } d > m(k) \\ 0 & \text{otherwise,} \end{cases}$$

MADE: Masked Autoencoder for Distribution Estimation

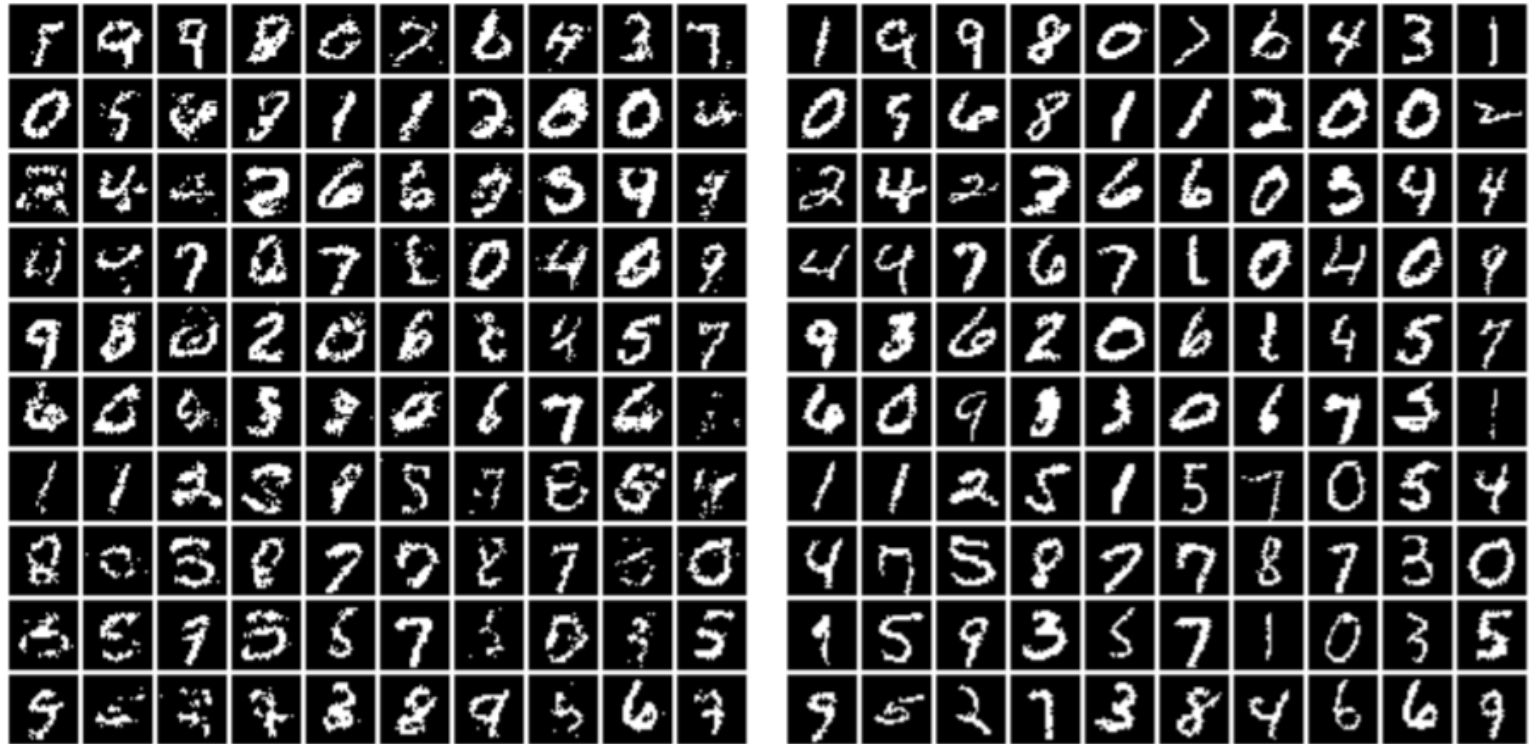


Figure 3. Left: Samples from a 2 hidden layer MADE. Right: Nearest neighbour in binarized MNIST.

Figure from MADE: Masked Autoencoder for Distribution Estimation, Germain et al '15

PixelCNN

Convolutional architecture, suitable for more complex image domains.

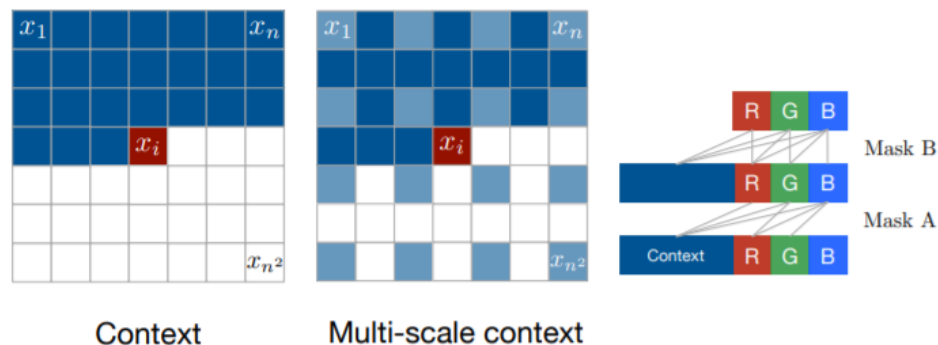


Figure 2. Left: To generate pixel x_i one conditions on all the previously generated pixels left and above of x_i . **Center:** To generate a pixel in the multi-scale case we can also condition on the subsampled image pixels (in light blue). **Right:** Diagram of the connectivity inside a masked convolution. In the first layer, each of the RGB channels is connected to previous channels and to the context, but is not connected to itself. In subsequent layers, the channels are also connected to themselves.

Figure from Pixel Recurrent
Neural Networks, van den
Oord '16

Obvious generalization – only implementation detail: channels are also generated “auto-regressively”.
 $p(x_i|x_{<i})$ is factorized as (Mask A)

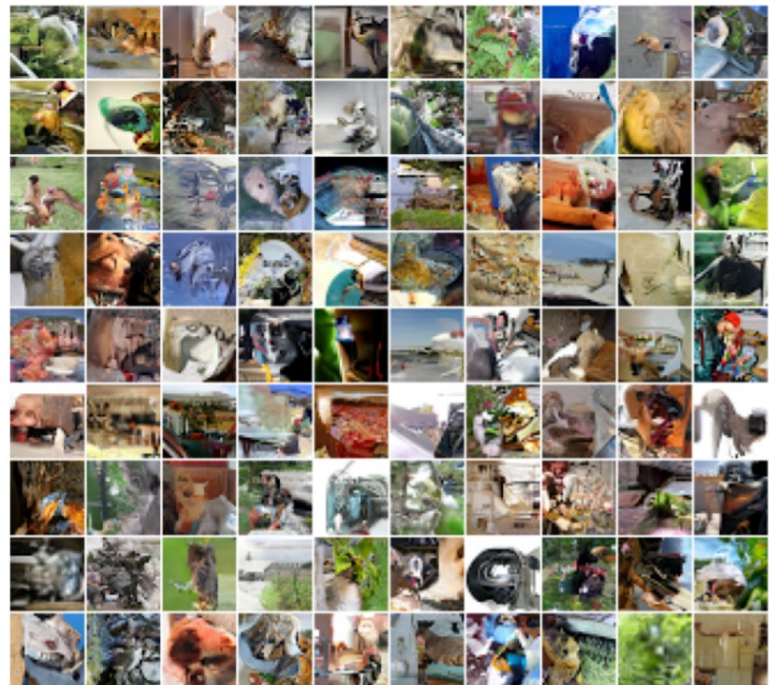
$$p(x_{i,R}|\mathbf{x}_{<i})p(x_{i,G}|\mathbf{x}_{<i}, x_{i,R})p(x_{i,B}|\mathbf{x}_{<i}, x_{i,R}, x_{i,G})$$

In upper layers, we use Mask B, in which value of channel can depend on value of same channel below. (This does the correct thing: e.g. G in layer two only depends on R in the input; if we used mask A, G would not depend on current input at all.)

PixelCNN



Figures from Pixel Recurrent
Neural Networks, van den
Oord '16



Recurrent neural networks

The problem with the previous approaches: number of parameters depend on total length.

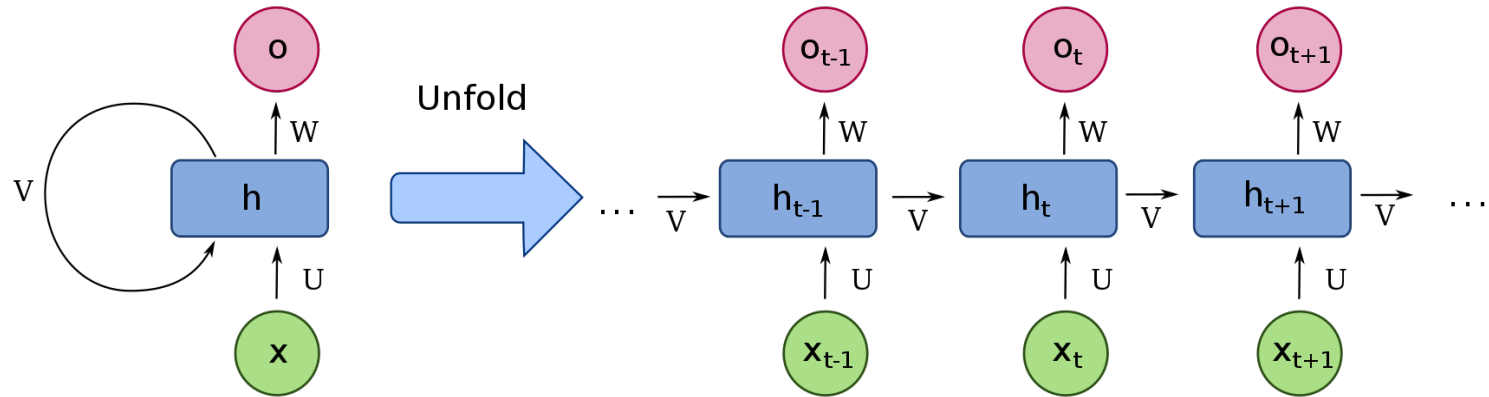
Recurrent neural networks are a way to *weight-tie* the parameters of an autoregressive model, s.t. it can be extended to arbitrary length sequences.

$$h_i = \tanh(W_{hh}h_{i-1} + W_{xh}x_i)$$

$$o_i = W_{hy}h_i$$

o_i specifies parameters for $p(x_i|x_{<i})$, e.g. $\text{softmax}(y_i)$

Recurrent neural networks



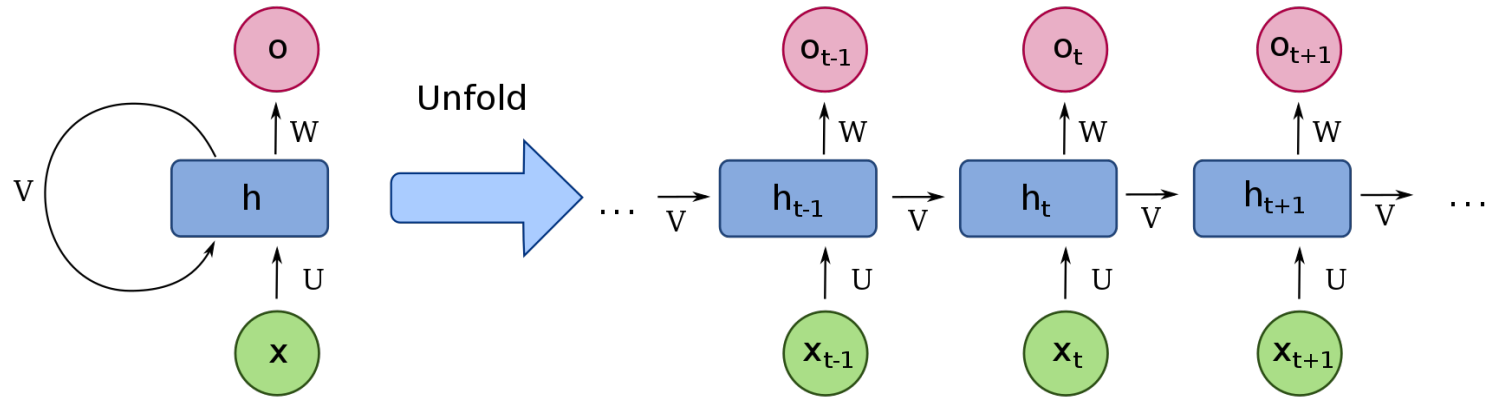
Obvious benefit: copious weight tying, number of params completely independent of length.

Drawbacks:

Training: backpropagation. Via unfolding equivalence above, same as calculating derivative of a length- t feedforward network.

Same problem as in very deep networks: *gradient explosion/vanishing!*

Recurrent neural networks



Obvious benefit: copious weight tying, number of params completely independent of length.

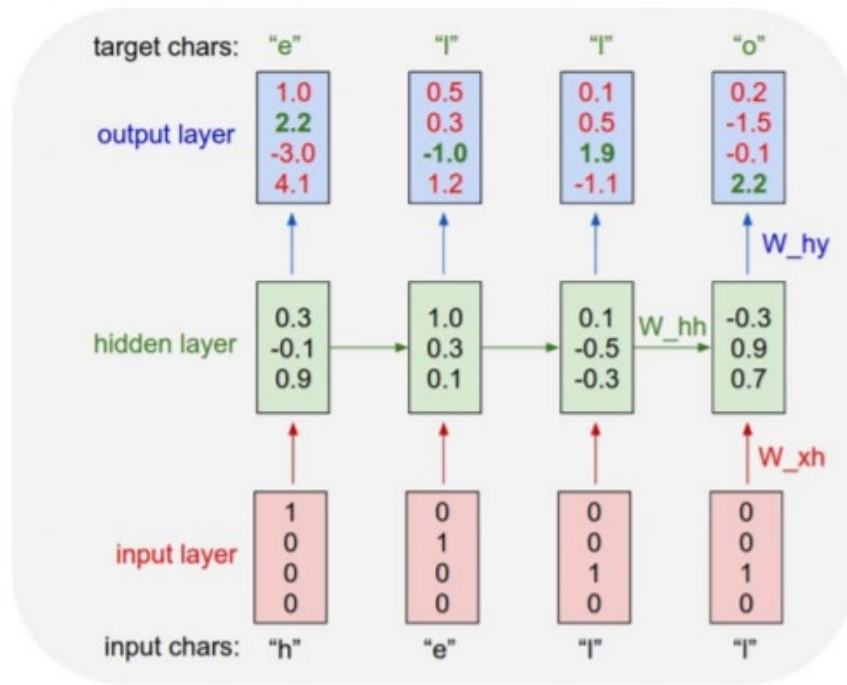
Drawbacks:

Training: *gradient explosion/vanishing!*

Likelihood evaluation: has to be done sequentially (no parallelization)

Recurrent neural networks

Example: Character-RNN (Figure from <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>)



$$h_i = \tanh(W_{hh}h_{i-1} + W_{xh}x_i)$$

$$o_i = W_{hy}h_i$$

x_i : characters ("e", "i", ...)

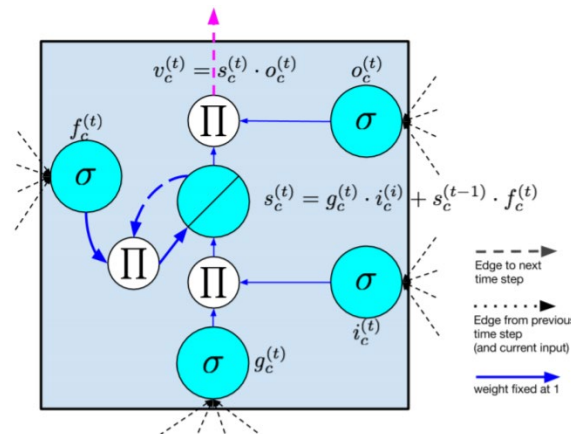
$p(x_i|x_{<i})$: softmax
with parameters o_i

LSTM (Long Short-Term Memory)

The main issue with training RNN's is long-term dependencies (and correspondingly exploding/vanishing gradients)

The main idea of **LSTM's** (*Hochreiter and Schmidhuber '97*): are gating mechanisms that try to control the flow of information from past.

In practice, they seem to suffer much less from gradient vanishing/explosion. (No good theoretical understanding.)



LSTM (Long Short-Term Memory)

Ingredients:

Input node: $g^{(t)} = \phi(W^{gx}x^{(t)} + W^{gh}h^{(t-1)} + b_g)$

Input gate: $i^{(t)} = \sigma(W^{ix}x^{(t)} + W^{ih}h^{(t-1)} + b_i)$

Forget gate: $f^{(t)} = \sigma(W^{fx}x^{(t)} + W^{fh}h^{(t-1)} + b_f)$

Output gate: $o^{(t)} = \sigma(W^{ox}x^{(t)} + W^{oh}h^{(t-1)} + b_o)$

Internal state: $s^{(t)} = g^{(t)} \odot i^{(i)} + s^{(t-1)} \odot f^{(t)}$

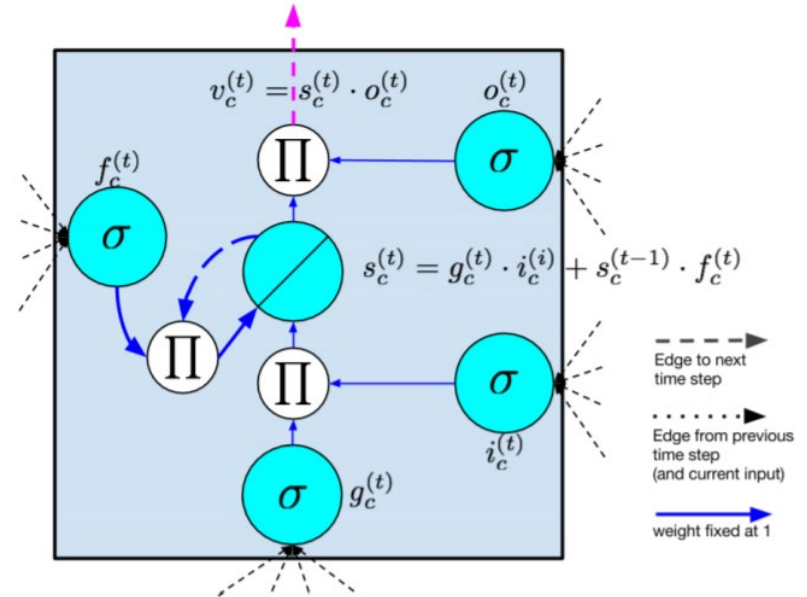
Hidden state: $h^{(t)} = \phi(s^{(t)}) \odot o^{(t)}$.

Input node will be “gated” by pointwise multiplication with input gate: how input “flows”

Will gate the “internal state”

How output “flows”

Combine gated version of prev. state w/ forget gate, along with gated input node.



RNN Examples

Example: Character-RNN (Figure from <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>)

Train 3-layer RNN with 512 hidden nodes on all the works of Shakespeare.
Then sample from the model:

KING LEAR: O, if you were a feeble sight, the courtesy of your law,
Your sight and several breath, will wear the gods
With his heads, and my hands are wonder'd at the deeds,
So drop upon your lordship's head, and your opinion
Shall be against your honour.

RNN Examples

Example: Character-RNN (Figure from <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>)

Train on Wikipedia. Then sample from the model:

Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25—21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict.

RNN Examples

when my cousin goes away there will
- (eg) med anche. 'bepestures the the
maine center of high credit
see Boring a. the accuracy is
pure in mist (saw) so lowest
bopes & cold mine's wine case
height. 4 Cees the gayer in
- style satet Jony in doing Te a
over & high earner. Tens. madp

Figure 11: Online handwriting samples generated by the prediction network. All samples are 700 timesteps long.

Figure from "Generating Sequences with RNNs", Graves 2014