10707
Deep Learning: Spring 2020

Andrej Risteski

Machine Learning Department

Lecture 18:
Invertible and autoregressive models

Part I: Invertible models

The idea behind GANs

Matching a distribution on images is hard because we don't have good measures of "**distance**" between images.
(Intuitively, two images could be very different in pixel space, while "**semantically**" being the same image.)

Why don't we simultaneously train a "distance" metric as we are training the model?

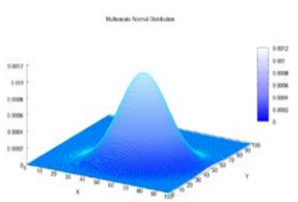
As a result, we will no longer be fitting the "maximum likelihood" model, but instead trying to learn some distribution close to the distribution of the input images in a learned metric.

This is (one of many) models which are "**likelihood-free**": we won't be able to explicitly write a likelihood for the model, but (importantly) we will efficiently be able to draw samples from the model!

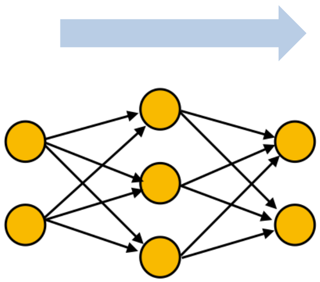
The GAN paradigm (Goodfellow et al. ‘14)

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

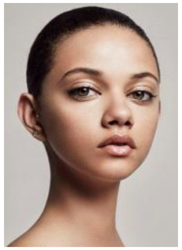
<u>Approach</u>: **Fit** distribution P_g parametrized by **neural network g**

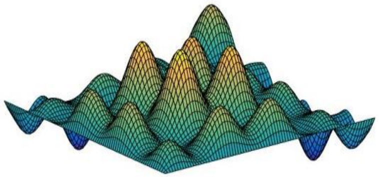


$Z \sim N(0, I_{k\times k})$



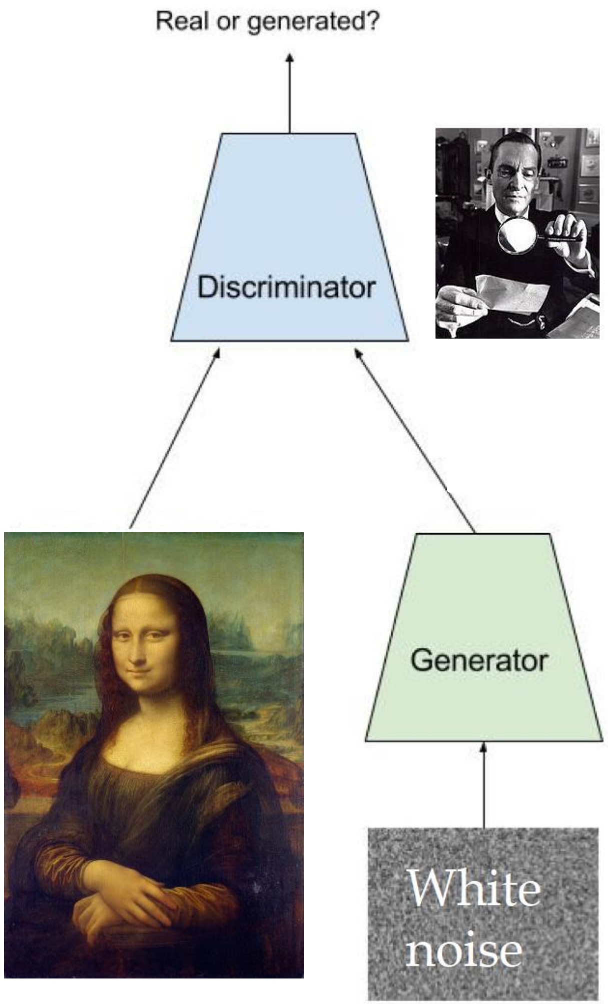
Neural network $g(\cdot)$





$X = g\,(Z)$

The GAN paradigm (Goodfellow et al. '14)



Game theoretic idea:

Generator trained to **fool** discriminator.

Discriminator trained to **beat** generator.

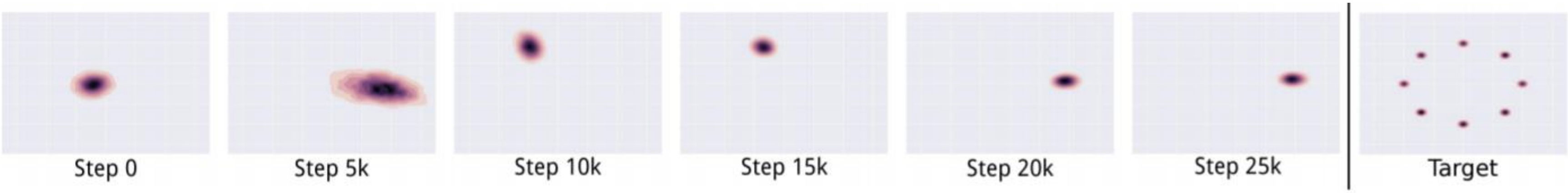
Common training problems

Unstable training: the problem is a min-max problem (also called saddle point problem) – typically optimization is much less stable than pure minimization.

Vanishing gradient: if the discriminator is too good, the generator gradients have a propensity to be small. (This is concerning, as to be taking gradients of the Wasserstein/JS/… objective, the discriminator needs to be optimal.)

Less of a problem with more modern GANs than with DC-GAN.

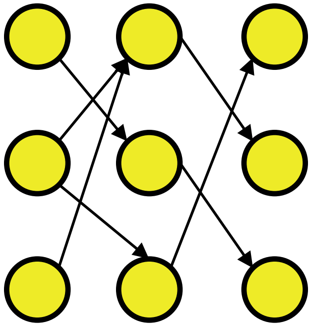
Mode collapse: the training only recovers some of the modes of the underlying distribution. **(NOT** clear if this is a statistical or algorithmic problem.)



Middle ground: "Invertible GANs"

Can we "marry" likelihood models w/ GANs?

Suppose g: $\mathbb{R}^d \to \mathbb{R}^d$ 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:

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

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

Middle ground: "Invertible GANs"

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

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

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

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 $f_\theta = f_1 \circ f_2 \circ \cdots f_L$, we have

Value of k-th layer

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

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 x d matrix takes $O(d^3)$ time – infeasible.

Choosing invertible transforms

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

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

<u>Efficient evaluation:</u> Determinant is diagonal (since $\frac{\partial f_\theta(x_i)}{\partial x_j} = 0$, for $i \neq j$), so $\det(J_x\left(f_\theta(x)\right) = \Pi_i \frac{\partial f_\theta(x_i)}{\partial x_i}$.

<u>Poor representational power:</u> 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 (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 $z \coloneqq f_\theta(x)$ into two sub-vectors, $z_{1:\frac{d}{2}}, z_{\frac{d}{2}+1,d}$ and set:

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

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

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

Choosing invertible transforms

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

Arbitrary s_θ, t_θ

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

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

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

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

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

If s_θ is say, a neural net, easy to evaluate and take derivatives.

Choosing invertible transforms

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

Figure from "*Density estimation using Real NVP*" by *Dinh et al '16*

Part II: Autoregressive models

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

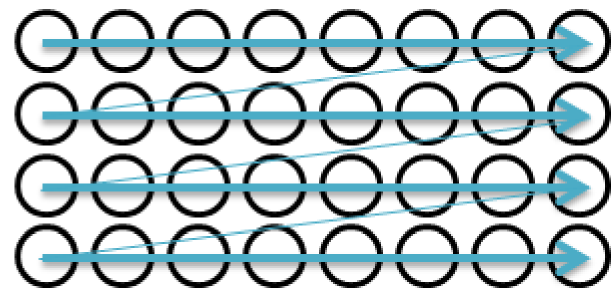
$$p(x_1, x_2, \dots, x_t) = p(x_i | x_{<i})$$

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

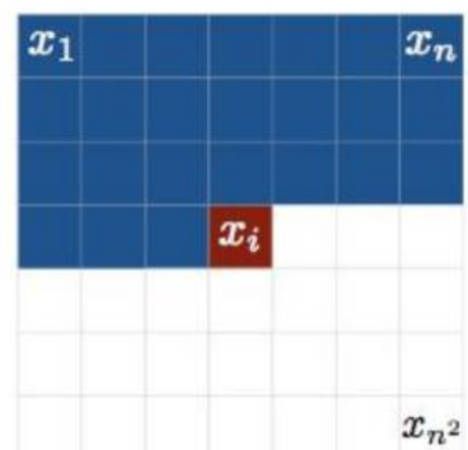
Sequential structure in data

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:



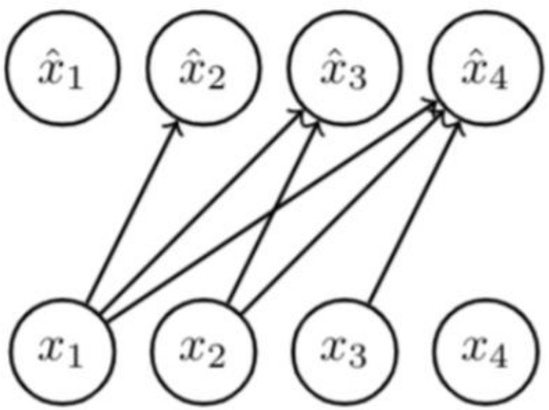
Fully visible sigmoid belief network

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

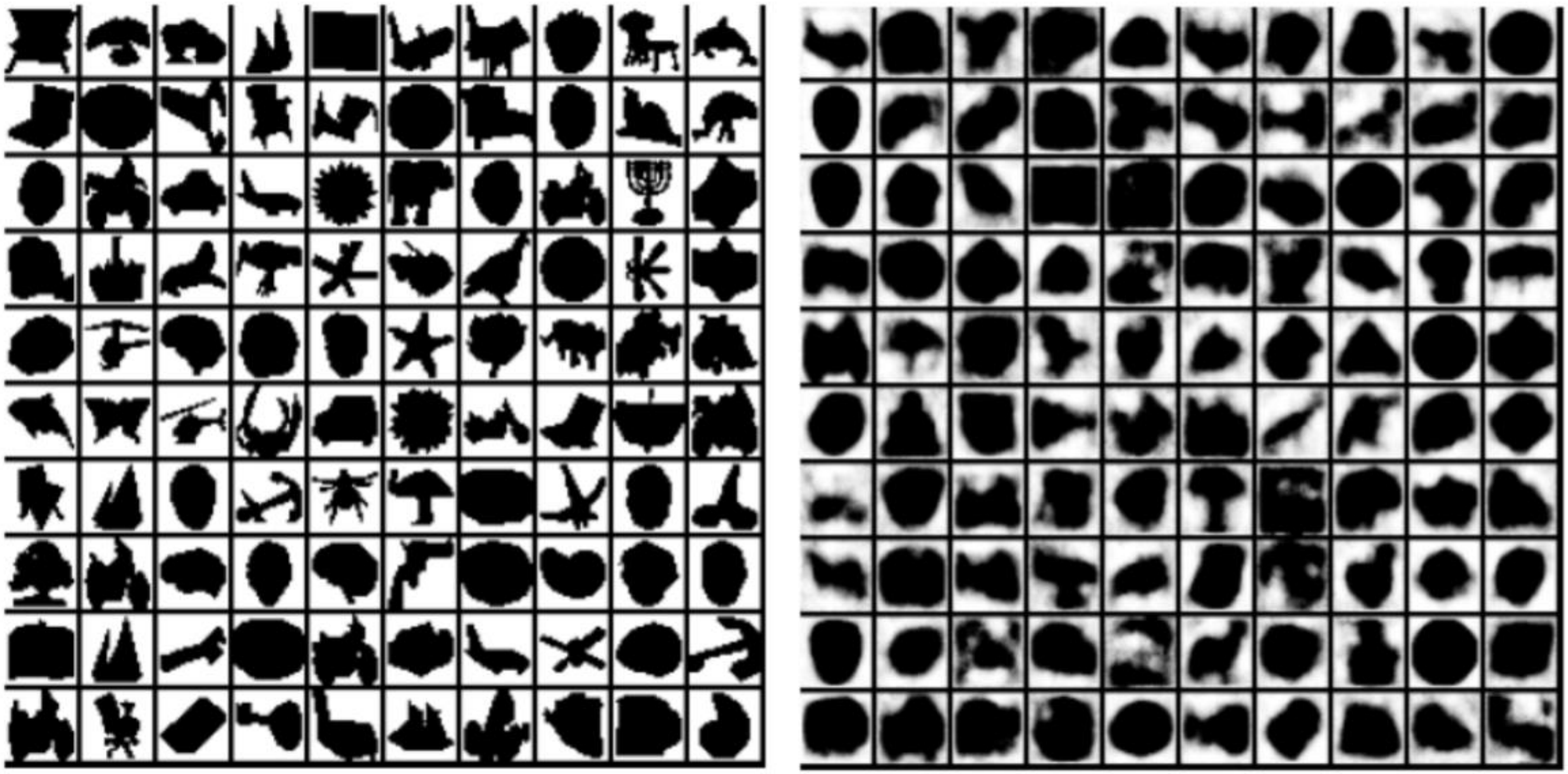


Figure from *Learning Deep Sigmoid Belief Networks with Data Augmentation, Gan et al' 2015*

NADE: Neural Autoregressive Density Estimation

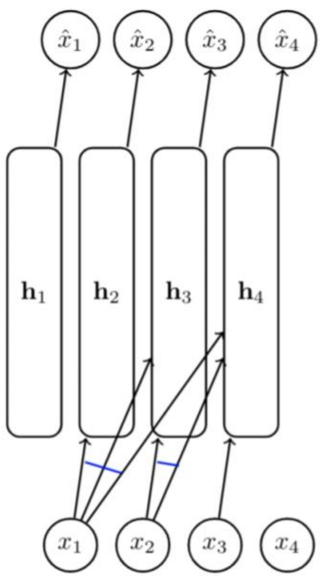
Let's assume data is binary (e.g. MNIST).

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

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

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

One-hidden layer net, sigmoid activation σ



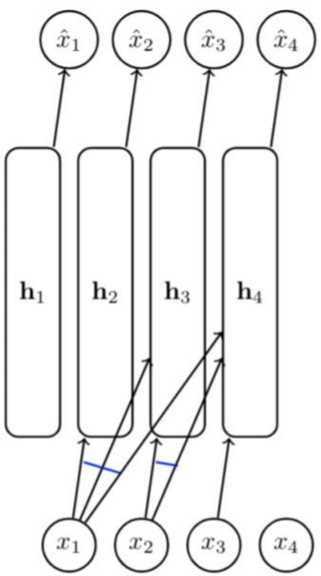
Sampling, training: same as before

NADE: Neural Autoregressive Density Estimation

Weight-tied variants:

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

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



e.g.

$$\mathbf{h}_2 = \sigma\left(\underbrace{\begin{pmatrix} \vdots \\ \mathbf{w}_1 \\ \vdots \end{pmatrix}}_{A_2} x_1\right) \quad \mathbf{h}_3 = \sigma\left(\underbrace{\begin{pmatrix} \vdots & \vdots \\ \mathbf{w}_1 & \mathbf{w}_2 \\ \vdots & \vdots \end{pmatrix}}_{A_3} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}\right) \quad \mathbf{h}_4 = \sigma\left(\underbrace{\begin{pmatrix} \vdots & \vdots & \vdots \\ \mathbf{w}_1 & \mathbf{w}_2 & \mathbf{w}_3 \\ \vdots & \vdots & \vdots \end{pmatrix}}_{A_3} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}\right)$$

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

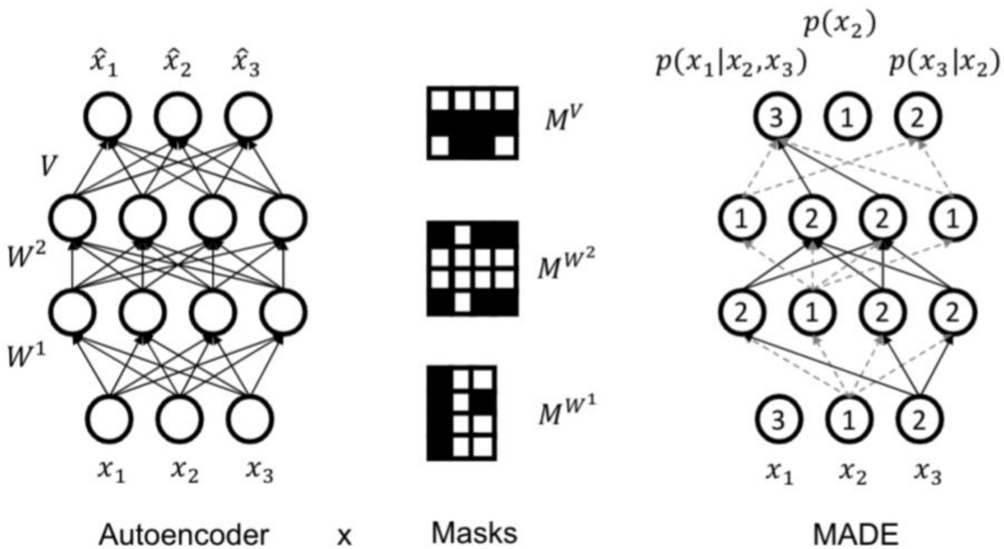
Strategy: turn an autoencoder into an autoregressive distribution modeler.

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, … $\hat{x}_2$ 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:

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

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.

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

MADE: Masked Autoencoder for Distribution Estimation

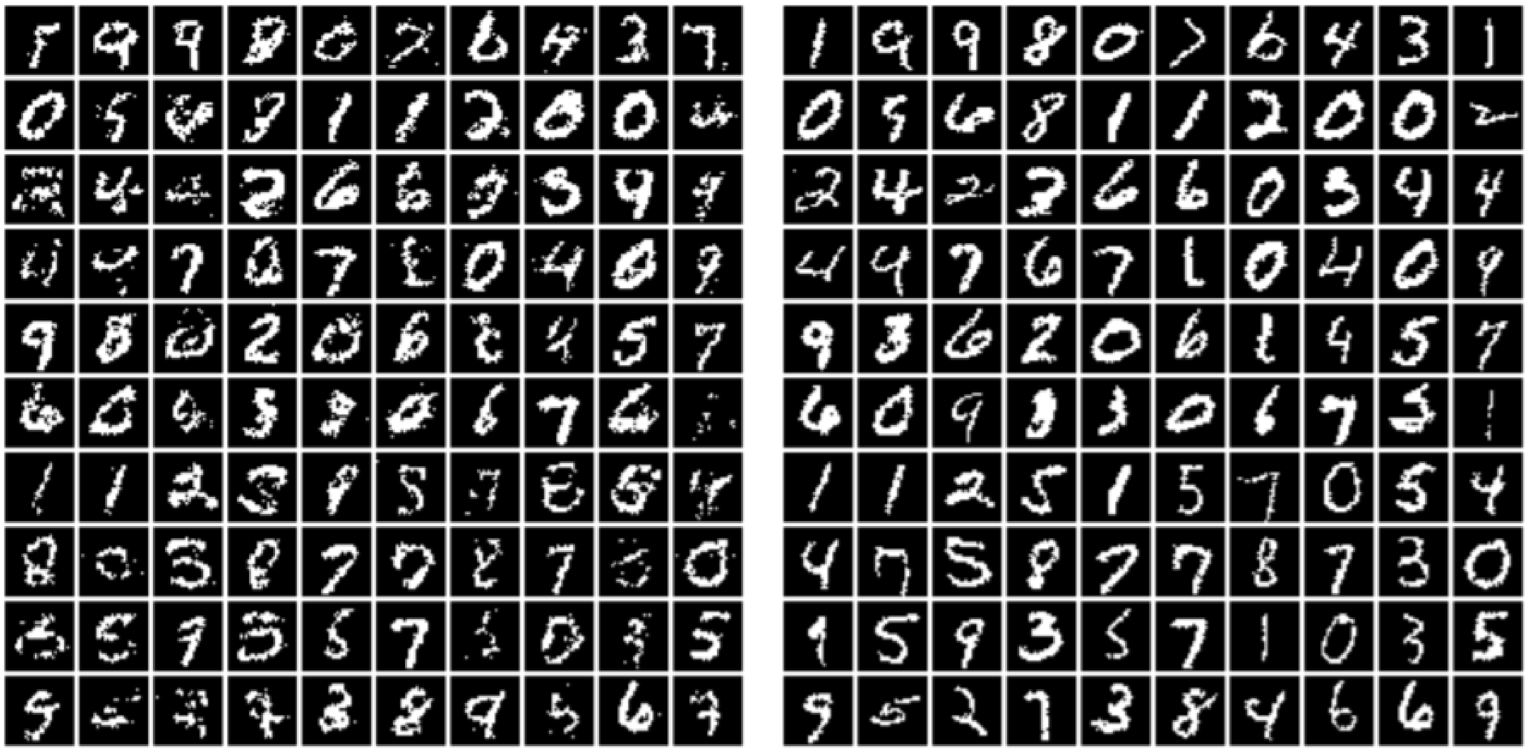


Figure 3. Left: Samples from a 2 hidden layer MADE. Right: Nearest neighbour in binarized MNIST.

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

PixelCNN

Convolutional architecture, suitable for more complex image domains.

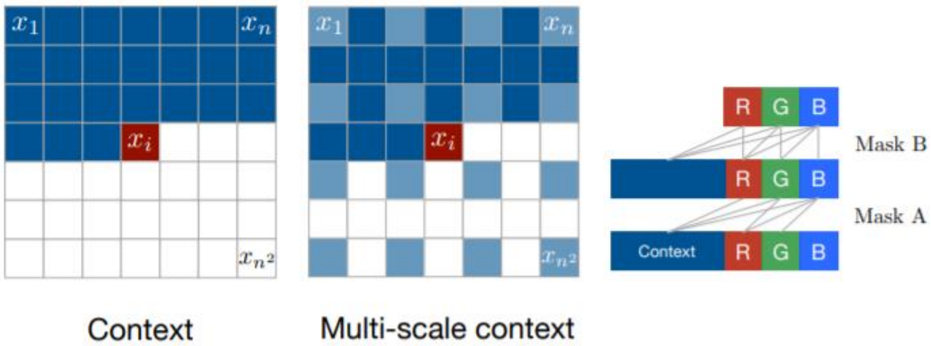


Figure 2. **Left**: To generate pixel x_i one conditions on all the previously generated pixels left and above of x_i. **Center**: To generate a pixel in the multi-scale case we can also condition on the subsampled image pixels (in light blue). **Right**: Diagram of the connectivity inside a masked convolution. 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.

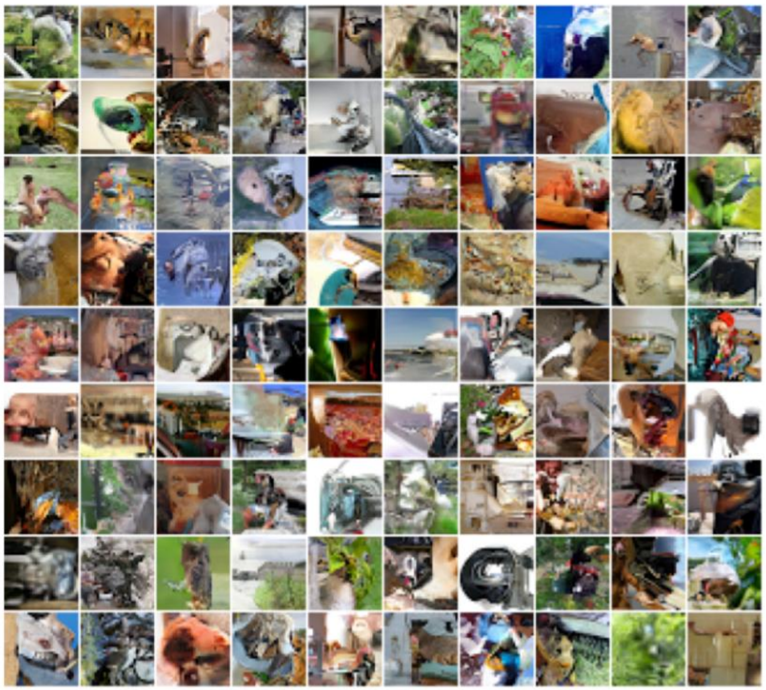
Figure from Pixel Recurrent Neural Networks, van den Oord '16

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

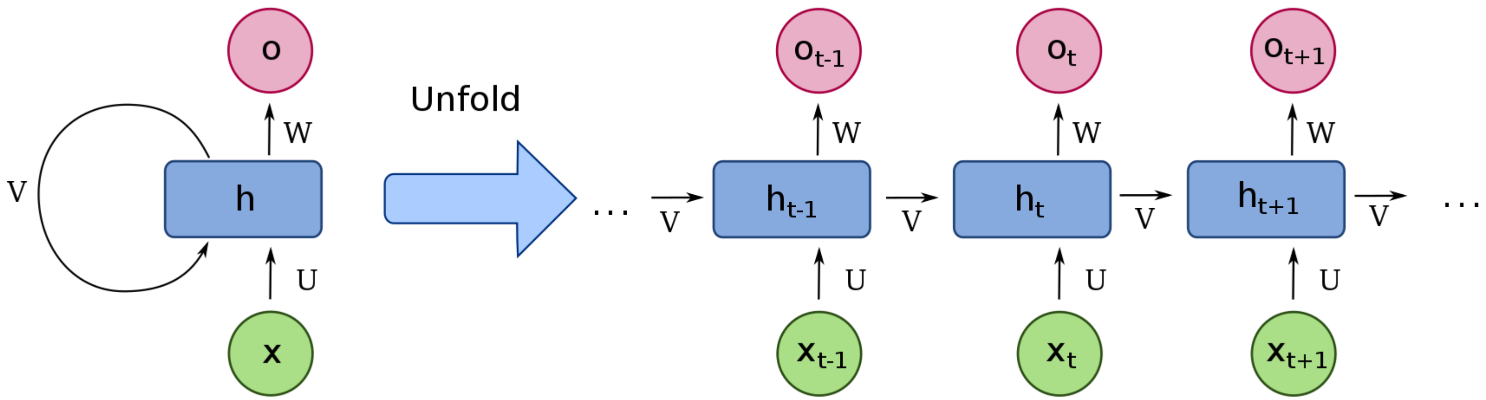
The problem with the previous approaches: number of parameters depend on total length.

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



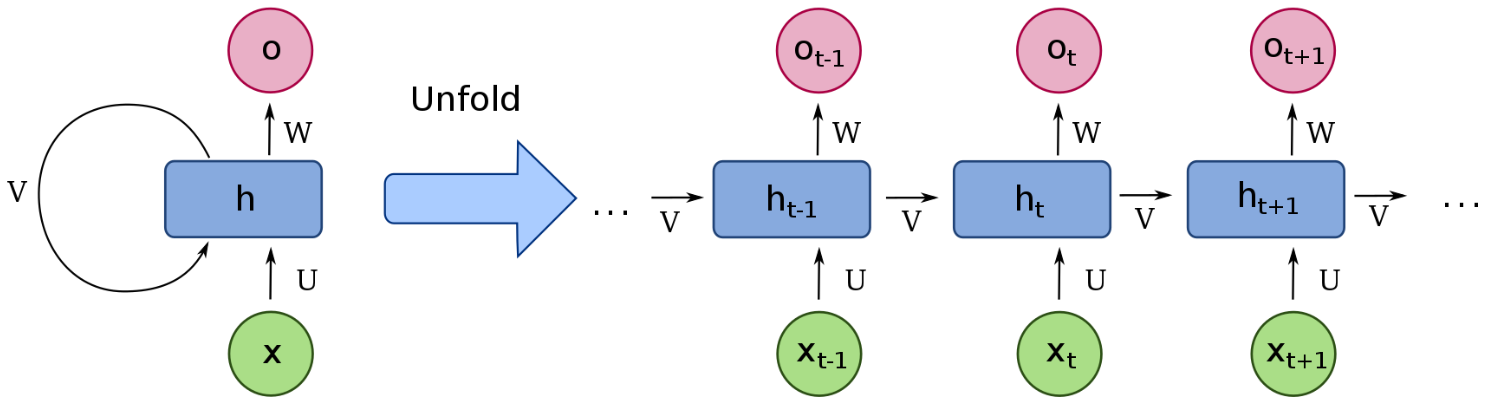
Obvious benefit: copious weight tieing, number of params completely independent of length.

Drawbacks:

<u>Training</u>: 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 tieing, number of params completely independent of length.

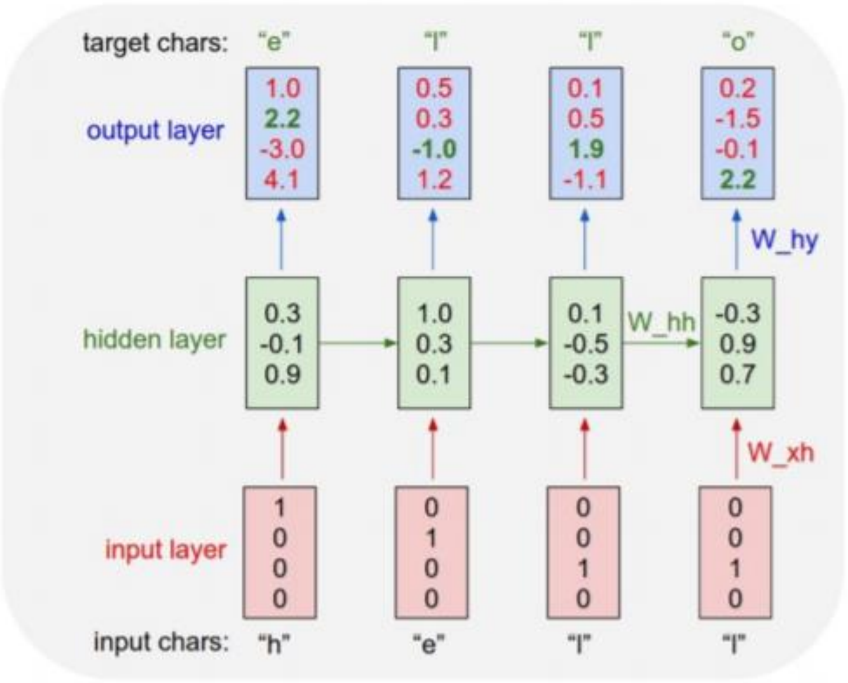
Drawbacks:

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

Recurrent neural networks

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.

Recurrent neural networks

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

Train on Wikipedia. Then sample from the model:

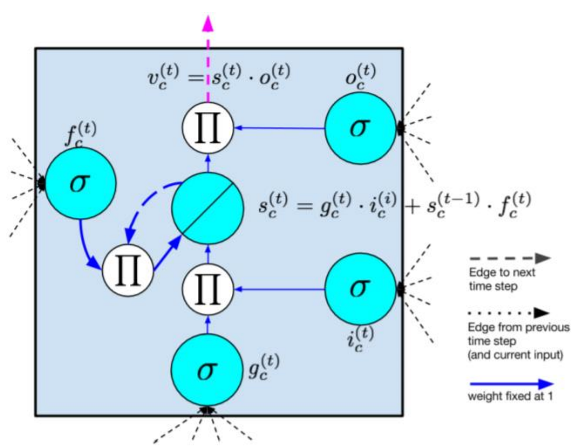
> Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be 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.

LSTM (Long Short-Term Memory)

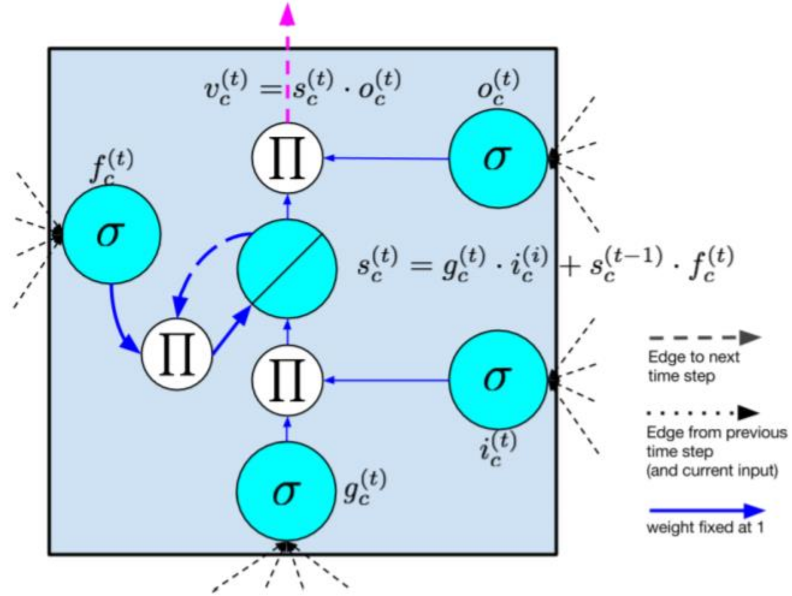
The main issue with training RNN’s is long-term dependencies (and correspondingly exploding/vanishing gradients)

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)



<u>Ingredients</u>:

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

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

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

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

Will gate the "internal state"

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

How output "flows"

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

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

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