G1: Autoregressive Models

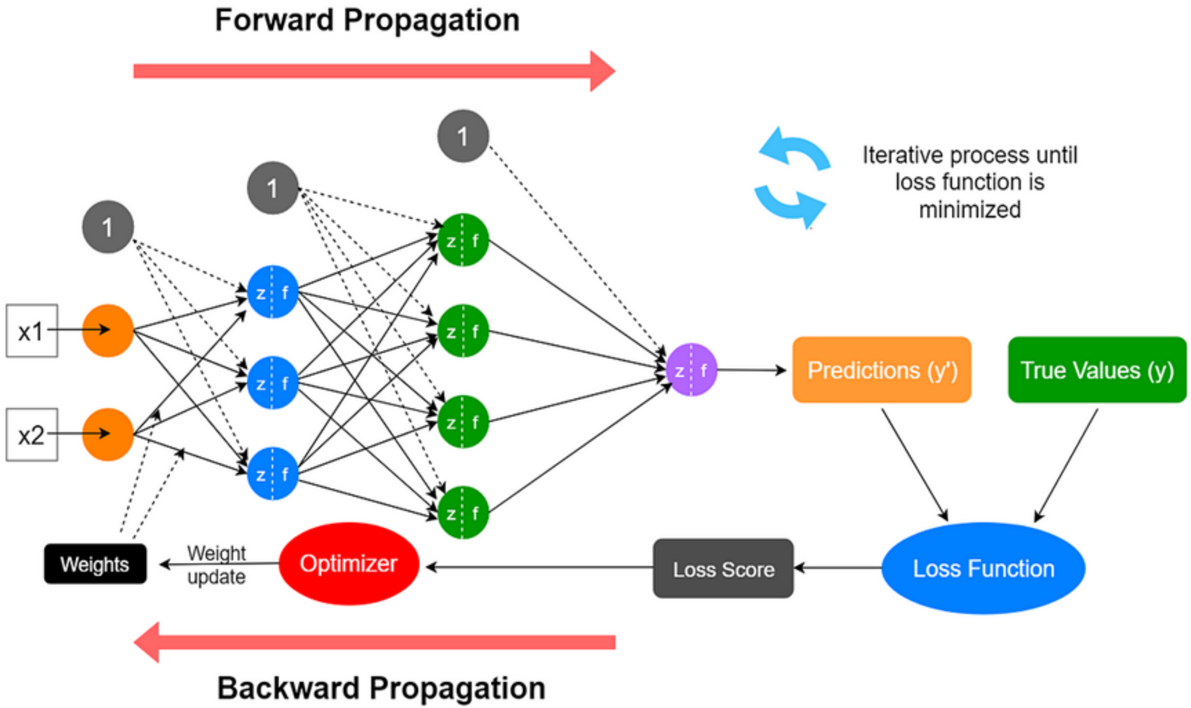
Content

- Concept of Autoregression
- Backward/Forward Pass

MADE:
- Autoencoders
- Distribution Estimation as Autoregression
- Masked Autoencoder
- Mask Generation
- Training Methods for MADE

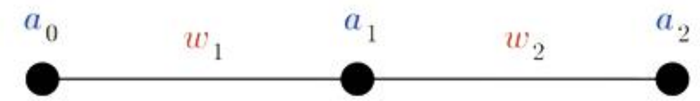
PixelCNN
- Intro & Convolutional layer
- Joint Distribution
- Masked Convolutions
- PixelCNN
- PixelCNN & PixelRNN structure
- PixelCNN vs PixelRNN

Backward/Forward Pass



https://medium.com/data-science-365/overview-of-a-neural-networks-learning-process-61690a502fa

Backpropagation: Simple Example



$$z_1 = a_0 w_1 + b_1$$

$$a_1 = A(z_1)$$

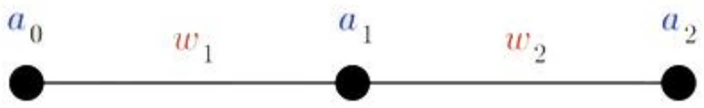
$$z_2 = a_1 w_2 + b_2$$

$$a_2 = A(z_2)$$

$$c = C(a_2, y)$$

$$\frac{\Delta c}{\Delta w_2}$$

Backpropagation (2)



$$z_1 = a_0 w_1 + b_1$$

$$a_1 = A(z_1)$$

$$z_2 = a_1 w_2 + b_2$$

$$a_2 = A(z_2)$$

$$c = C(a_2, y)$$

$$\frac{\partial c}{\partial w_2} = \frac{\partial z_2}{\partial w_2} \times \frac{\partial a_2}{\partial z_2} \times \frac{\partial c}{\partial a_2}$$

Motivation for autoregressive models

- Goal: estimate underlying distribution of data
- Predict future values in sequential data by using past values (autoregression)

MADE

Masked Autoencoder for Distribution Estimation

Motivation for MADE

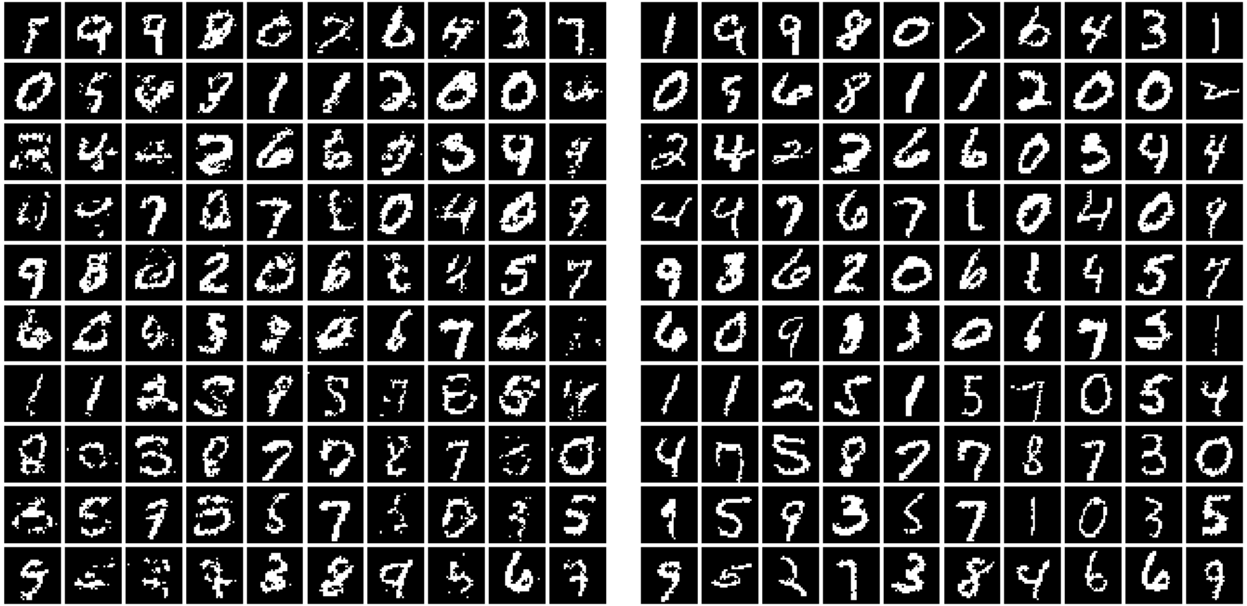
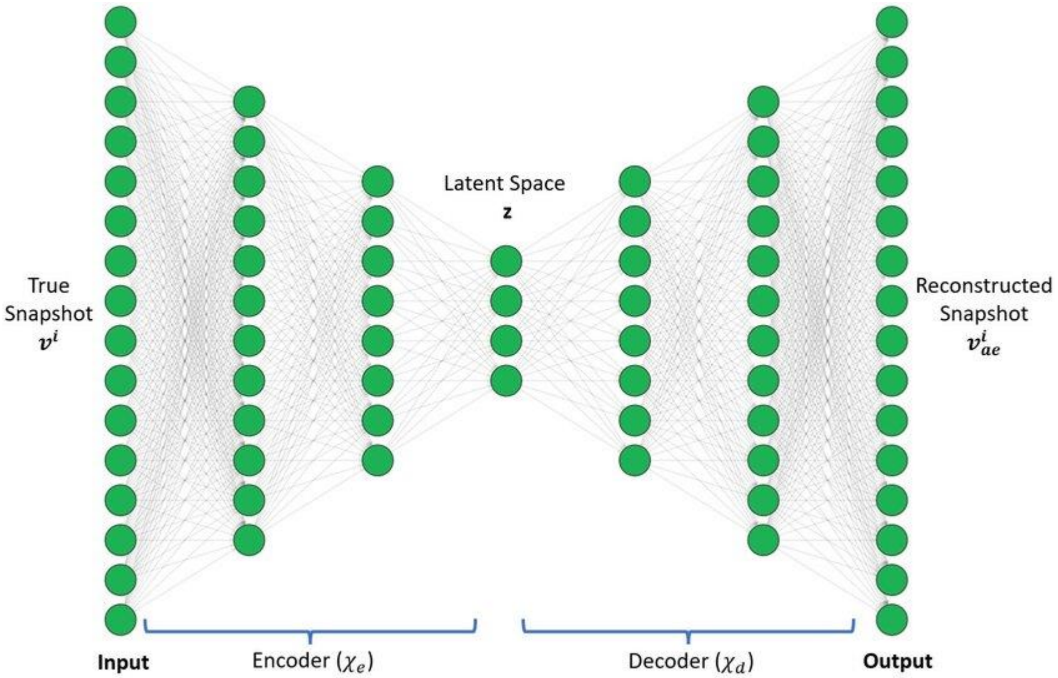


Figure 3. Left: Samples from a 2 hidden layer MADE. Right: Nearest neighbour in binarized MNIST.

Traditional autoencoders



https://www.researchgate.net/figure/MLP-Autoencoder-architecture_fig4_373266879

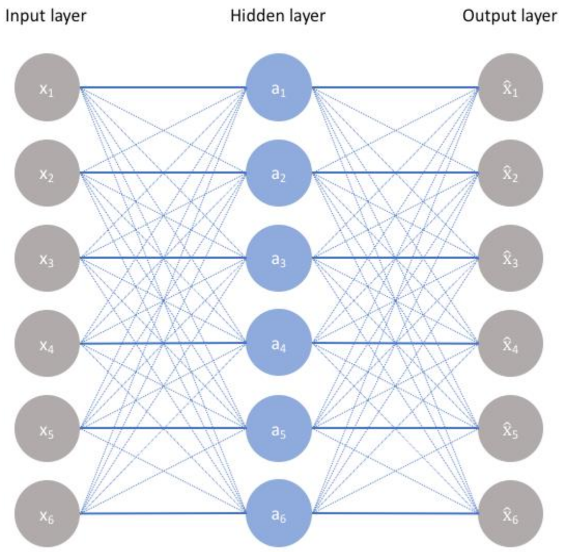
Goal: reconstruct input and learn compressed representation

Traditional Autoencoders(2)

- Effective at capturing and representing data distributions implicitly

- but only by applying constraints

- Possible constraints:
- Less connections
- Less nodes
- Add noise



https://www.jeremyjordan.me/autoencoders/

Constrain autoencoder to output data distribution(MADE)

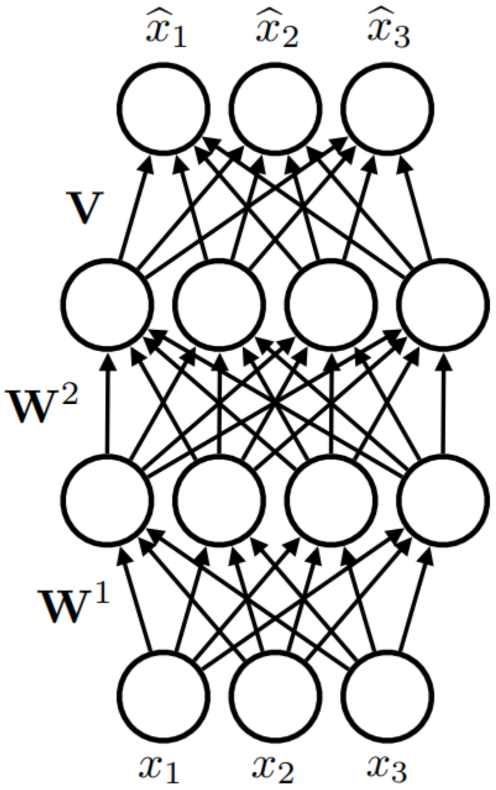
Architecture of MADE

- MADE is generally built like an Autoencoder

- Autoencoders learn a compressed representation *h(x)* of input *x* to reconstruct *x* as closely as possible:

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

- b, c — bias vectors
- W, V — weight matrices
- g, sigm — activation functions
- $\hat{x}$ — output (reconstructed value)

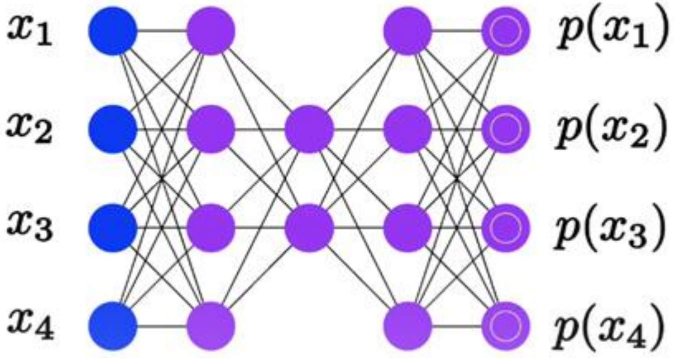


First attempt

- output is probabilty
  → joint distribution p(x)

- but this uses i.i.d assumption

$$p(\mathbf{x}) = p(x_1)p(x_2)p(x_3)p(x_4)$$



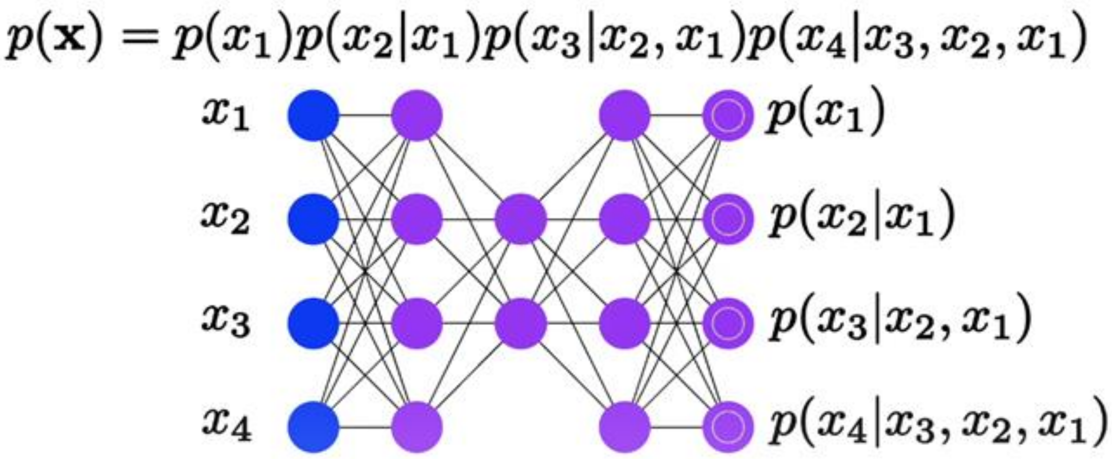
https://towardsdatascience.com/made-masked-autoencoder-for-distribution-estimation-fc95aaca8467

i.i.d. assumption

- **i.i.d.** means ***independent identically distributed***
- **Independent:** one sample does not influence others
- **Identically distributed:** all samples come from the same probability distribution

In our networks **not** the case

(almost) MADE structure



https://towardsdatascience.com/made-masked-autoencoder-for-distribution-estimation-fc95aaca8467

Ordering of data is arbitrary(order agnostic training)

MADE

- Decomposing distribution, into the product of its nested conditionals (product rule)

$$p(x) = \prod_{d=1}^{D} p(x_d | x_{<d}), \qquad where \ \ x_{<d} = [x_1, \dots, x_{d-1}]^T$$

- the binary autoencoder uses the cross-entropy loss

$$\ell(\mathbf{x}) = \sum_{d=1}^{D} -x_d \log \widehat{x}_d - (1-x_d) \log(1-\widehat{x}_d)$$

Using the cross-entropy loss like before:

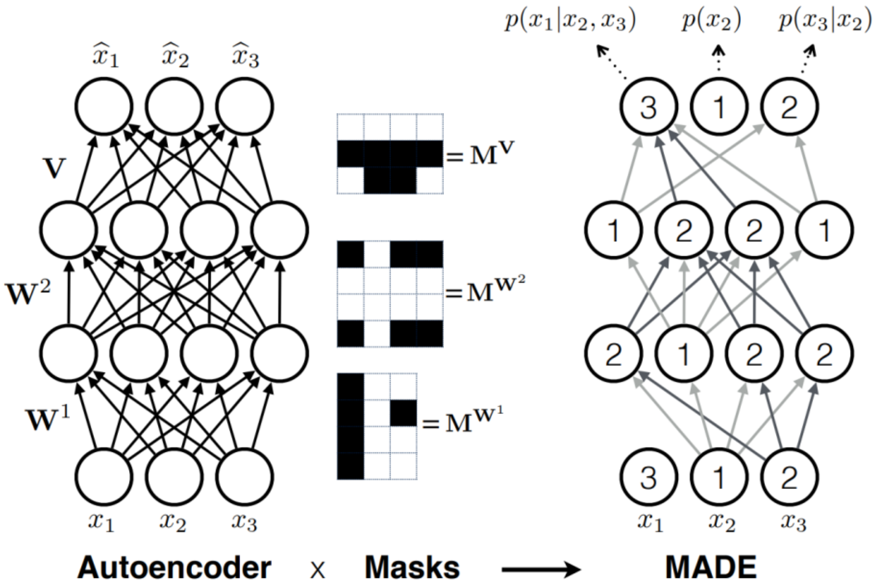
$$\ell(\mathbf{x}) = \sum_{d=1}^{D} -x_d \log \widehat{x}_d - (1-x_d) \log(1-\widehat{x}_d)$$

$$defining\ \ p(x_d = 1 \mid x_{<d}) = \hat{x}_d\ \ and\ \ p(x_d = 0 \mid x_{<d}) = 1 - \hat{x}_d,$$

$$We\ get\ a\ valid\ negative\ log\ likelihood\ l(x)$$

$$\begin{aligned} \ell(x) &= \sum_{d=1}^{D} -x_d \log p(x_d = 1 \mid x_{<d}) - (1 - x_d) \log p(x_d = 0 \mid x_{<d}) \\ &= \sum_{d=1}^{D} - \log p(x_d \mid x_{<d}) \\ &= - \log p(x) \end{aligned}$$

Masked Autoencoder: Intuition



Goal: Erase all non-autoregressive connections (logically)

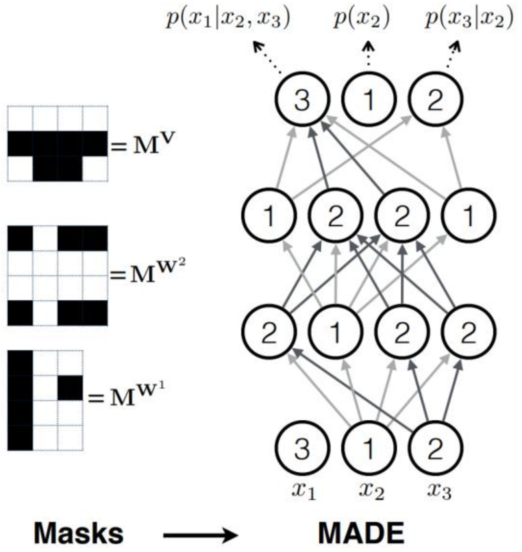
Masked Autoencoder

- Apply masks to enforce autoregressive property

- Elementwise multiplication with a binary mask matrix, setting entries to 0 for desired removal:

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

- $\mathrm{M}^{\mathrm{W}}, \mathrm{M}^{\mathrm{V}}$: Masks for W & V



Mask generation: Constraints

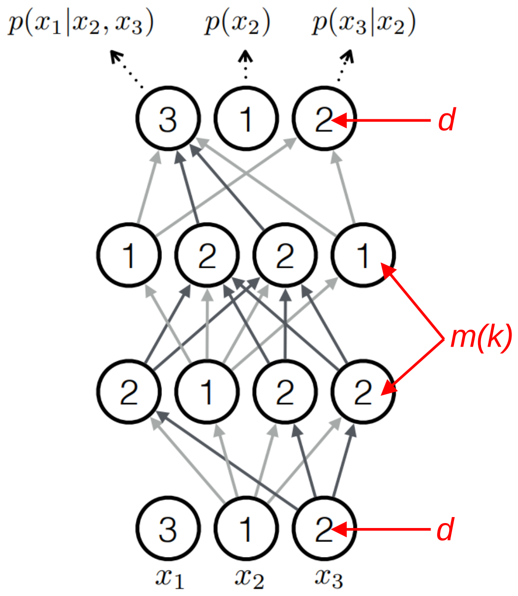
- $m \triangleq$ *integer between* 1 & $D-1$
- $k \triangleq$ *hidden unit* $\Rightarrow m(k) \Leftrightarrow \max(\#\textit{input connections to unit } k)$

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

- The d^{th} output unit should only be connected to $x_{<\text{d}}$

$\Rightarrow$ output weights can only connect to units with $d > m(k)$

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



MADE

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

Extension to Deep Networks

- Network with l layers
- Layer 0,…,l-1

$$M^{\mathbf{W}^l}_{k',k} = 1_{m^l(k') \geq m^{l-1}(k)} = \begin{cases} 1 & \text{if } m^l(k') \geq m^{l-1}(k) \\ 0 & \text{otherwise.} \end{cases}$$

- Output layer

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

Reference:

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

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

Training Methods for MADE

Order-Agnostic Training

- Training on all orderings of the input
- Missing values can be easily computed
- Ensemble of models with different orderings and averaging

Connectivity-Agnostic Training

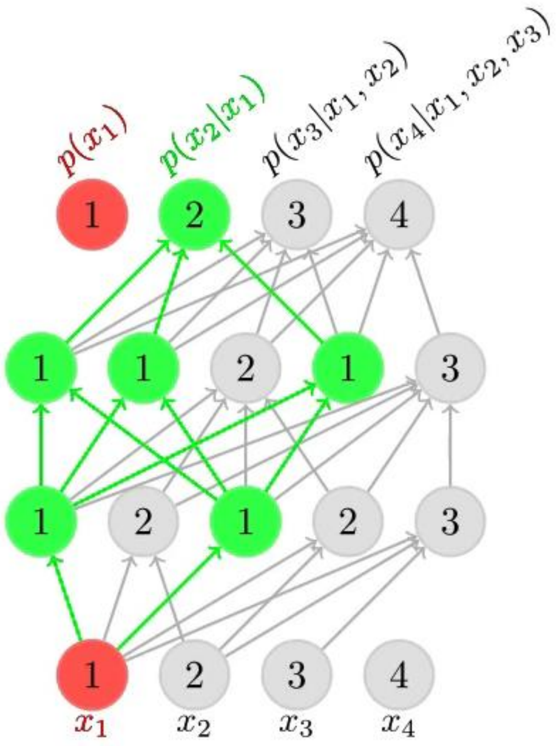
- Connectivity pattern is randomly sampled for each training example
- NN needs to be more robust → better results

Using both results in an overall better performance

MADE: Sampling

1. Sample value from x_1
2. Feed value into network & compute next pixel value
3. Feed x_1 & x_2 into Network & compute next pixel value
4. …

- Sampling is sequential → takes more time
- Training is parallel → more efficient



https://youtu.be/lNW8T0W-xeE?si=LgUjyPHeKFvK0xkc

PixelCNN & PixelRNN

Pixel Convolutional Neural Networks & Pixel Recurrent Neural Networks

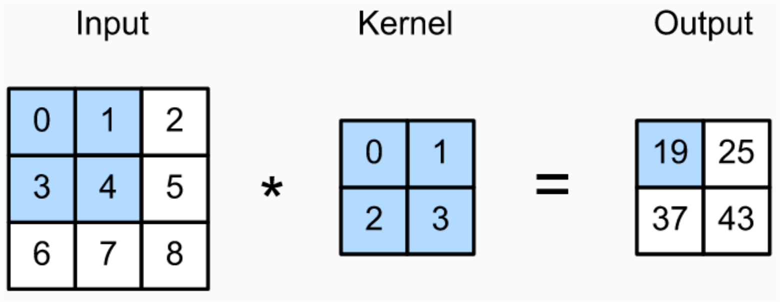
PixelCNN & PixelRNN: Intro

- Goal: model the distribution of natural images
- models should be tractable and scalable
- Usage: Image completion or sample new images

Figure 1. Image completions sampled from a PixelRNN.

General Concept of Convolution

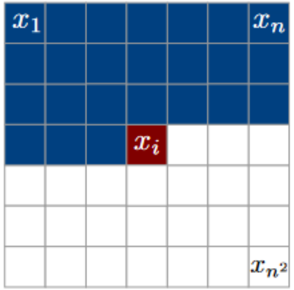
- Filter/ Kernel: (weight) matrix used for convolution
- Stride: step width of the filter
- Padding: preserve spatial dimensions by adding zeros around the input



https://bios691-deep-learning-r.netlify.app/class/04-class/

PixelCNN/PixelRNN: joint distribution

- Pixels of $n \times n$ image taken left-to-right, top-to-bottom
- Joint distribution:



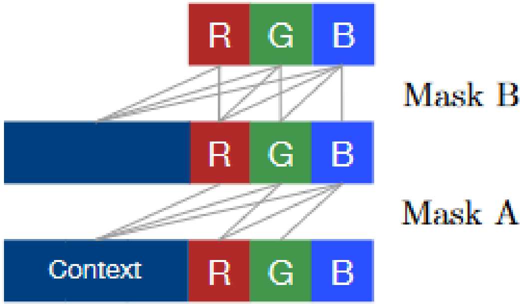
$$p(x) = \prod_{i=1}^{n^2} p(x_i | x_1, \dots, x_{i-1})$$

$$p(x) = p\left(x_{i,R} \middle| x_{<i}\right) p\left(x_{i,G} \middle| x_{<i}, x_{i,R}\right) p\left(x_{i,B} \middle| x_{<i}, x_{i,R}, x_{i,G}\right)$$

Each pixel depends on previous pixels (also in different channels)

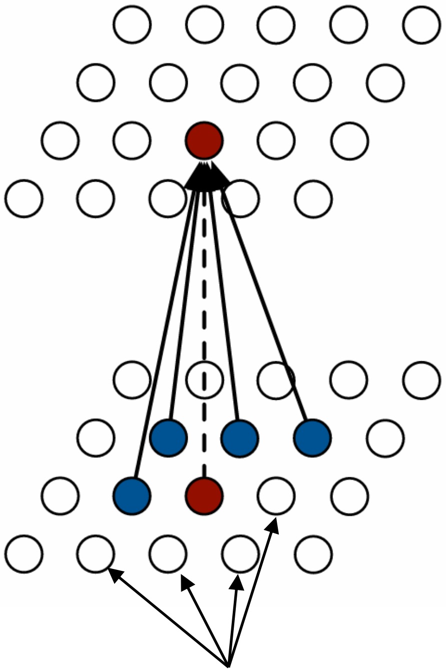
Masked Convolutions

- Masks are applied to convolutions to avoid seeing future context
- Pixel dependencies are kept
- Two types of masks (A & B)



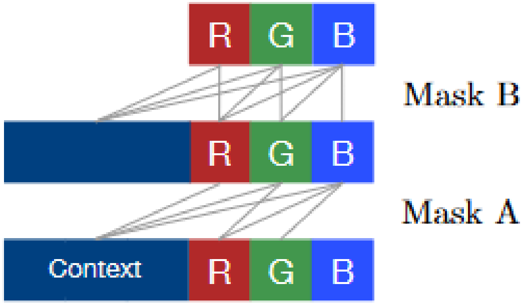
Masked Convolutions: Mask A

- Applied to first convolution filter
- Restricts connections to previous pixels and colours of the current pixels that have already been predicted

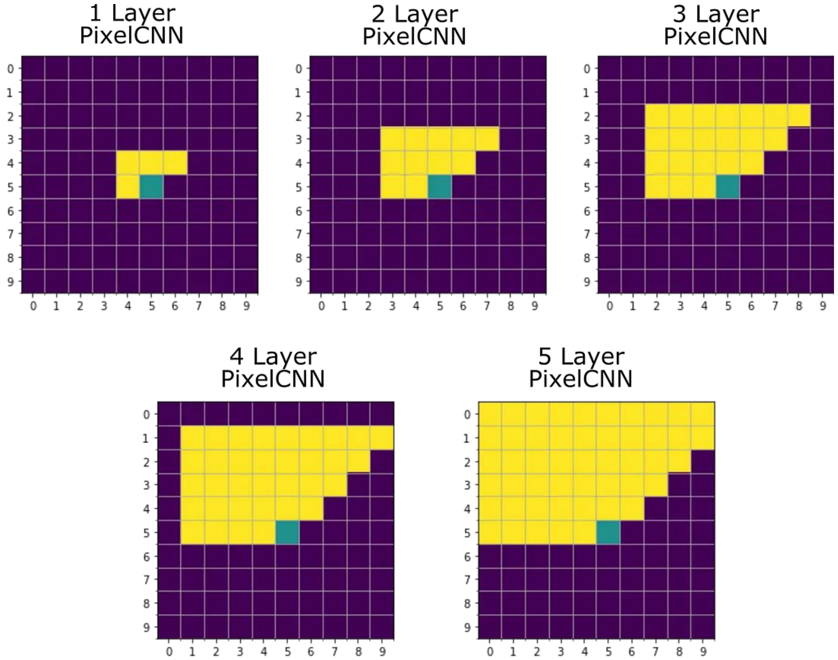


Masked Convolutions: Mask B

- Applied to subsequent input-to-state convolutional transitions
- Allows a connection from a colour to itself

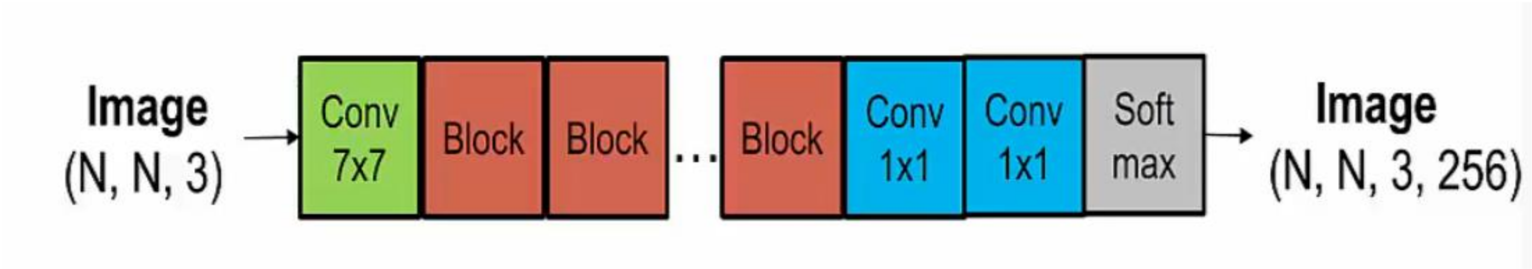


PixelCNN



https://miro.medium.com/v2/resize:fit:1100/format:webp/1*V0V1blD6mdGkPmYDede3dw.png

Structure: PixelCNN/PixelRNN



Typical architecture

https://neuroverse0.wordpress.com/2020/08/11/pixelrnn-gated-pixelcnn-and-pixelcnn/

PixelCNN: residual block

- Input map is added to the output map
- Goal: increase convergence and propagate signals more directly

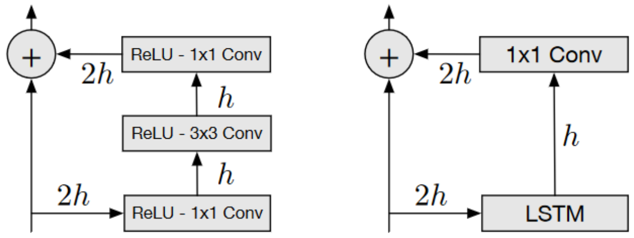
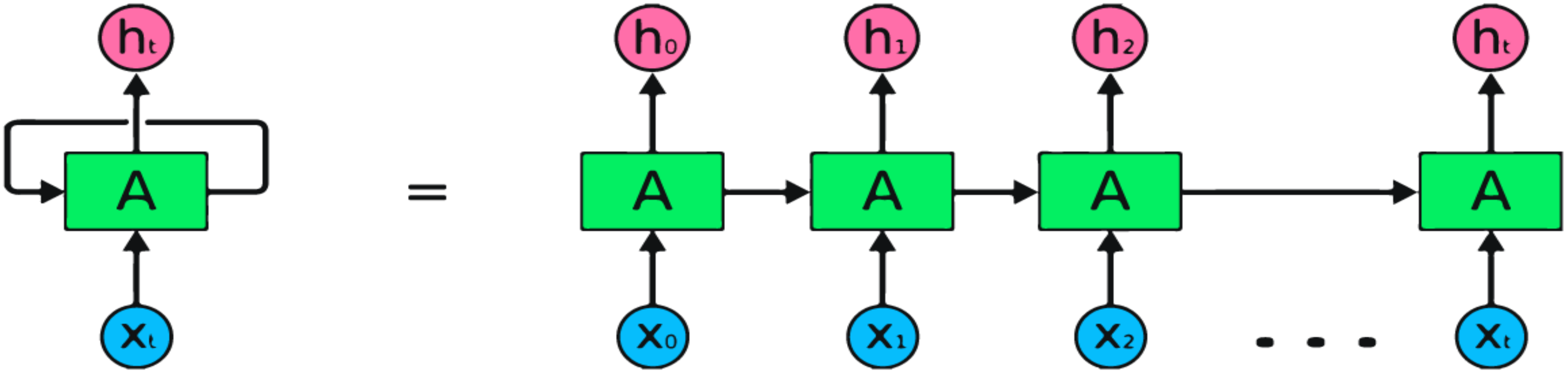


Figure 5. Residual blocks for a PixelCNN (left) and PixelRNNs.

Recurrent Neural Networks (RNN)

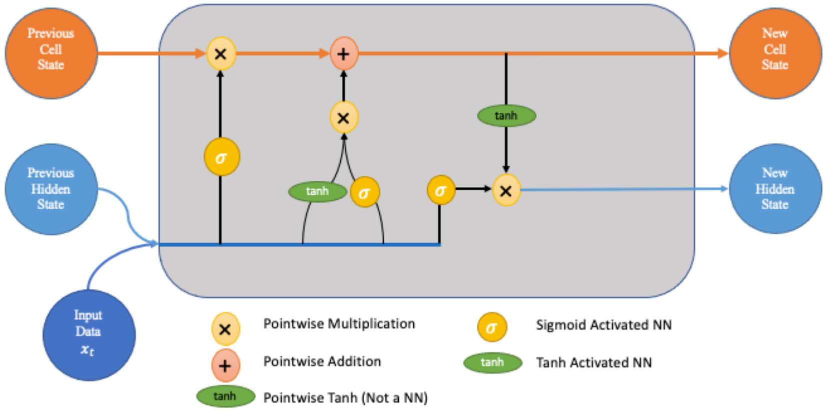


https://images.datacamp.com/image/upload/v1647442110/image2_ysmali.png

LSTM(Long short-term memory)

- A special kind of Recurrent Neural Network (RNN) that are designed to learn long-term dependencies in sequence data

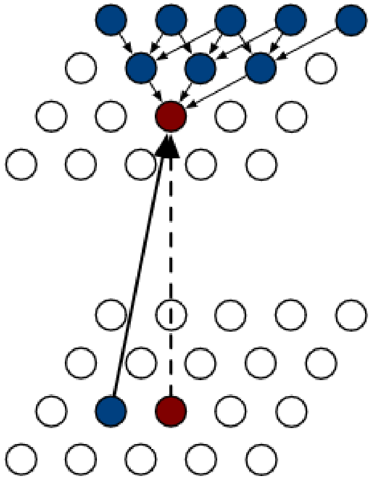
- Components of LSTM:
    - Input-to-state
    - State-to-state
    - Four gates (output, input, forget, content)



https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9

PixelRNN: Row LSTM

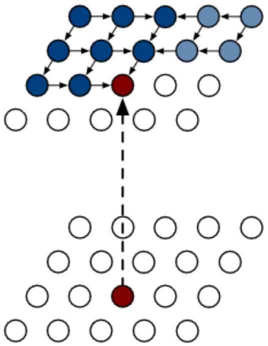
- Unidirectional layer
- Processes image row by row
- Triangular receptive field (missing available context)



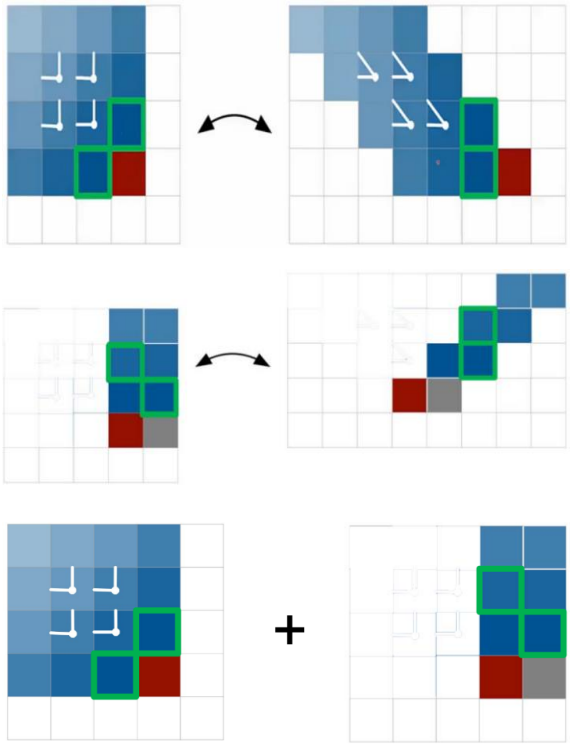
Row LSTM

PixelRNN: Diagonal BiLSTM

- Captures entire available context
- Scans image in diagonal from the top corners



Diagonal BiLSTM



Final state-to-state component

https://neuroverse0.wordpress.com/2020/08/11/pixelrnn-gated-pixelcnn-and-pixelcnn/

Structure: PixelCNN vs PixelRNN

PixelCNN	Row LSTM	Diagonal BiLSTM
7×7 conv mask A		
Multiple residual blocks: (see fig 5)		
Conv 3×3 mask B	Row LSTM i-s: 3×1 mask B s-s: 3×1 no mask	Diagonal BiLSTM i-s: 1×1 mask B s-s: 1×2 no mask
ReLU followed by 1×1 conv, mask B (2 layers)		
256-way Softmax for each RGB color (Natural images) or Sigmoid (MNIST)		

Table 1. Details of the architectures. In the LSTM architectures i-s and s-s stand for input-state and state-state convolutions.

PixelCNN/RNN: output

- 256-way Softmax for each RGB colour
- Softmax model discrete probability distribution
- Advantages: distribution mass inside [0,255] ; predicted distributions are meaningful

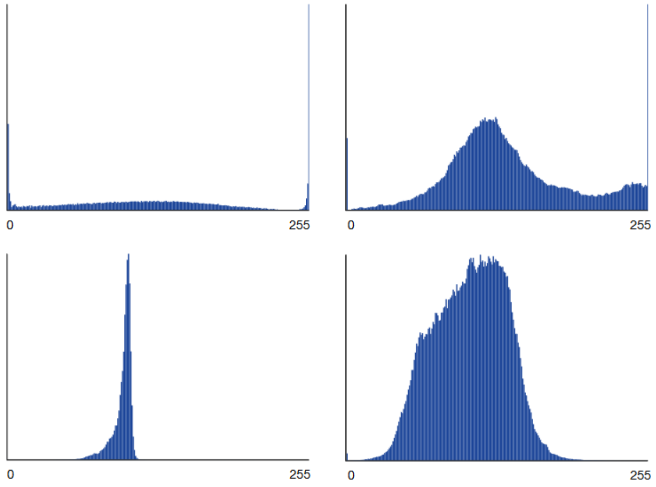


Figure 6. Example softmax activations from the model. The top left shows the distribution of the first pixel red value (first value to sample).

PixelCNN vs PixelRNN

PixelRNN

- Uses LSTMs to capture dependencies
- Computational more intensive in training
  → each state needs to be computed sequentially
- Captures entire available context

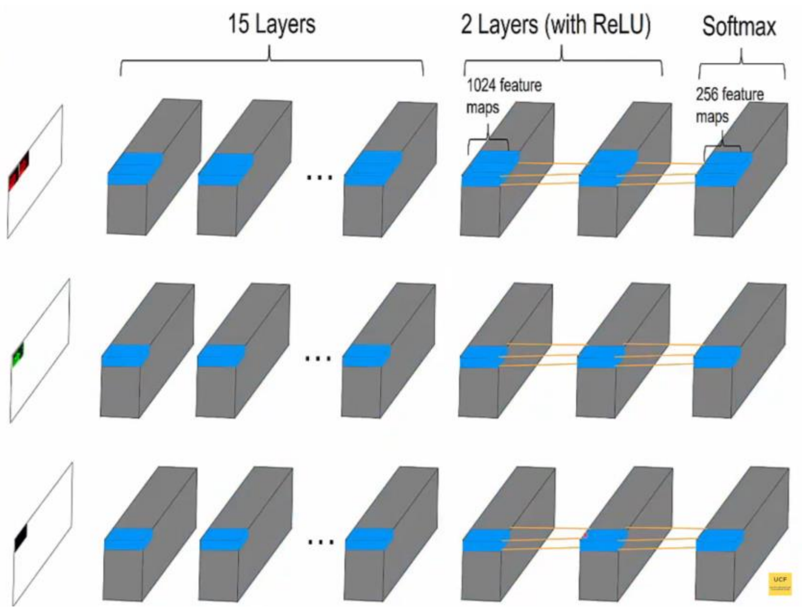
PixelCNN

- Only uses convolutional layers
- Computational more efficient in training/evaluation
  → computes features for all pixel positions at once
- Limited receptive field (not all available context captured)

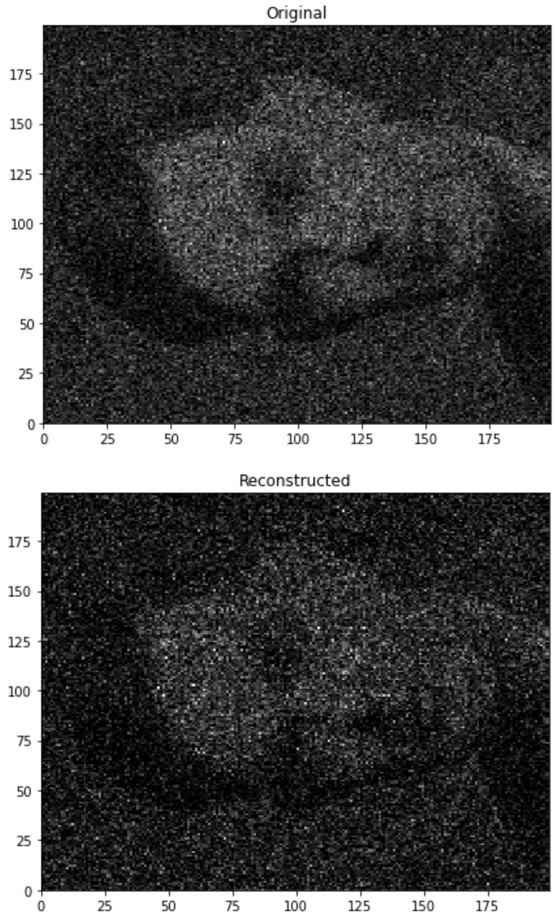
PixelCNN: Sampling

1. Sample value from x_1
2. Feed value into network & compute next pixel value
3. Feed x_1 & x_2 into Network & compute next pixel value
4. …

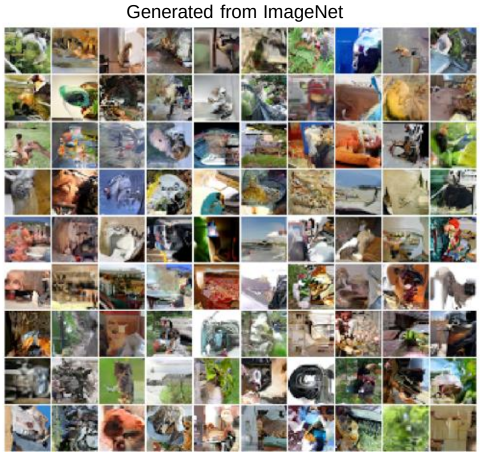
- Sampling is sequential → takes more time
- Training is parallel (CNN) → more efficient



https://youtu.be/-FFveGrG46w?si=BOHiIvvzY-GnDJgU



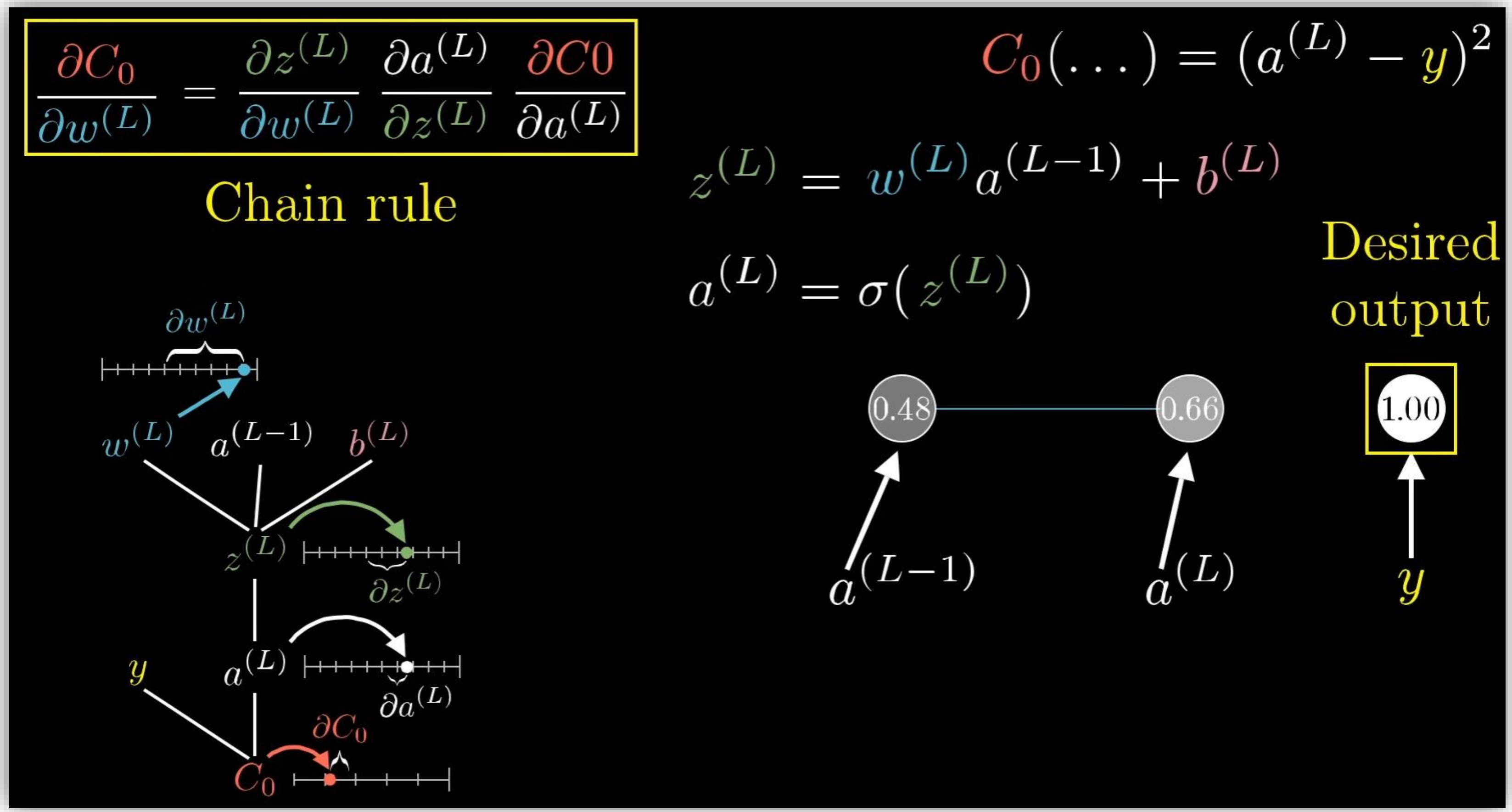
MADE vs PixelRNN/CNN



Appendix

Sources

- Pixel Recurrent Neural Networks – Google DeepMind
- MADE: Masked Autoencoder for Distribution Estimation - Mathieu Germain
- https://medium.com
- https://miro.medium.com
- https://youtu.be/hfMk-kjRv4c?si=UGt55n13H9mIYcJa - Sebastian Lauge
- https://youtu.be/tIeHLnjs5U8?si=5IEHnfWKft9AT6lP – 3Blue1Brown
- MADE - blog by Kapil Sachdeva
- https://bios691-deep-learning-r.netlify.app/class/04-class
- https://neuroverse0.wordpress.com/2020/08/11/pixelrnn-gated-pixelcnn-and-pixelcnn
- https://images.datacamp.com/image/upload/v1647442110/image2_ysmali.png
- https://towardsdatascience.com/lstm-networks-a-detailed-explanation-8fae6aefc7f9
- https://www.jeremyjordan.me/autoencoders
- https://www.researchgate.net/figure/MLP-Autoencoder-architecture_fig4_373266879



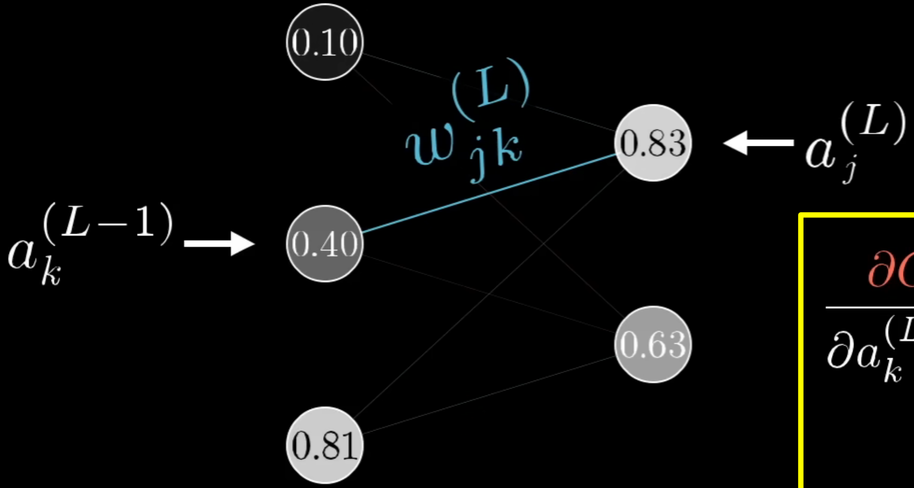
$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$
Chain rule
$C_0(\ldots) = (a^{(L)} - y)^2$
$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$
$a^{(L)} = \sigma(z^{(L)})$
Desired output
$\partial w^{(L)}$
$w^{(L)}$
$a^{(L-1)}$
$b^{(L)}$
$z^{(L)}$
$\partial z^{(L)}$
y
$a^{(L)}$
$\partial a^{(L)}$
∂C_0
C_0
0.48
0.66
1.00
$a^{(L-1)}$
$a^{(L)}$
y

$$\boxed{\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}}$$

$$z_j^{(L)} = \cdots + w_{jk}^{(L)} a_k^{(L-1)} + \cdots$$

$$a_j^{(L)} = \sigma(z_j^{(L)})$$

$$C_0 = \sum_{j=0}^{n_L-1} (a_j^{(L)} - y_j)^2$$



$$\boxed{\frac{\partial C_0}{\partial a_k^{(L-1)}} = \underbrace{\sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}}_{\text{Sum over layer L}}}$$