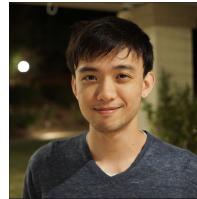
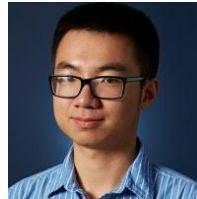


CS294-158 Deep Unsupervised Learning

Lecture 1c: Likelihood Models I: Autoregressive Models



Pieter Abbeel, Peter Chen, Jonathan Ho, Aravind Srinivas
UC Berkeley

Outline

- Motivation
- Simple generative models
 - Histograms
 - Illustration of failures on high-dimensional data
- Modern neural autoregressive models
 - Training procedure
 - Architecture

Likelihood-based models

How do we solve these problems?

- Generating data: synthesizing images, videos, speech, text
- Compressing data: constructing efficient codes
- Anomaly detection

Likelihood-based models: estimate p_{data} from samples $x^{(1)}, \dots, x^{(n)} \sim p_{\text{data}}(x)$

Learns a distribution p that allows:

- Computing $p(x)$ for arbitrary x
- Sampling $x \sim p(x)$

Today: **discrete** data

Desiderata

We want to estimate distributions of **complex, high-dimensional data**

- A 128x128x3 image lies in a ~50,000-dimensional space

We also want computational and statistical efficiency

- Efficient training and model representation
- Expressiveness and generalization
- Sampling quality and speed
- Compression rate and speed

Estimating frequencies by counting

Recall: the goal is to estimate p_{data} from samples $x^{(1)}, \dots, x^{(n)} \sim p_{\text{data}}(x)$

Suppose the samples take on values in a finite set $\{1, \dots, k\}$

The model: a **histogram**

- (Redundantly) described by k nonnegative numbers: p_1, \dots, p_k
- To train this model: count frequencies

$$p_i = (\# \text{ times } i \text{ appears in the dataset}) / (\# \text{ points in the dataset})$$

At runtime

Inference (querying p_i for arbitrary i): simply a lookup into the array p_1, \dots, p_k

Sampling (lookup into the inverse cumulative distribution function)

1. From the model probabilities p_1, \dots, p_k , compute the cumulative distribution

$$F_i = p_1 + \dots + p_i \quad \text{for all } i \in \{1, \dots, k\}$$

2. Draw a uniform random number $u \sim [0, 1]$
3. Return the smallest i such that $u \leq F_i$

Are we done?

Failure in high dimensions

No, because of the **curse of dimensionality**. Counting fails when there are too many bins.

- (Binary) MNIST: 28x28 images, each pixel in {0, 1}
- There are $2^{784} \approx 10^{236}$ probabilities to estimate
- Any reasonable training set covers only a tiny fraction of this
- Each image influences only one parameter. No generalization whatsoever!

Solution: function approximation. Instead of storing each probability, store a parameterized function $p_\theta(x)$

Likelihood-based generative models

Recall: the goal is to **estimate p_{data}** from $x^{(1)}, \dots, x^{(n)} \sim p_{\text{data}}(x)$

Now we introduce **function approximation**: learn θ so that $p_\theta(x) \approx p_{\text{data}}(x)$.

- How do we design function approximators to effectively represent complex joint distributions over x , yet remain easy to train?
- There will be many choices for model design, each with different tradeoffs and different compatibility criteria.

Designing the model and the training procedure go hand-in-hand.

Fitting distributions

- Given data $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$ sampled from a “true” distribution p_{data}
- Set up a model class: a set of parameterized distributions p_{θ}
- Pose a search problem over parameters

$$\arg \min_{\theta} \text{loss}(\theta, \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)})$$

- Want the loss function + search procedure to:
 - Work with large datasets (n is large, say millions of training examples)
 - Yield θ such that p_{θ} matches p_{data} — i.e. the training algorithm *works*. Think of the loss as a distance between distributions.
 - Note that the training procedure can only see the empirical data distribution, not the true data distribution: we want the model to generalize.

Maximum likelihood

- Maximum likelihood: given a dataset $\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}$, find θ by solving the optimization problem

$$\arg \min_{\theta} \text{loss}(\theta, \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(n)}) = \frac{1}{n} \sum_{i=1}^n -\log p_{\theta}(\mathbf{x}^{(i)})$$

- Statistics tells us that if the model family is expressive enough and if enough data is given, then solving the maximum likelihood problem will yield parameters that generate the data
- Equivalent to minimizing KL divergence between the empirical data distribution and the model

$$\hat{p}_{\text{data}}(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}[\mathbf{x} = \mathbf{x}^{(i)}]$$

$$\text{KL}(\hat{p}_{\text{data}} \| p_{\theta}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}}[-\log p_{\theta}(\mathbf{x})] - H(\hat{p}_{\text{data}})$$

Stochastic gradient descent

Maximum likelihood is an optimization problem. How do we solve it?

Stochastic gradient descent (SGD).

- SGD minimizes expectations: for f a differentiable function of θ , it solves

$$\arg \min_{\theta} \mathbb{E}[f(\theta)]$$

- With maximum likelihood, the optimization problem is

$$\arg \min_{\theta} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [-\log p_{\theta}(\mathbf{x})]$$

- **Why maximum likelihood + SGD?** It works with large datasets and is compatible with neural networks.

Designing the model

- Key requirement for maximum likelihood + SGD: efficiently compute $\log p(x)$ and its gradient
- We will choose models p_θ to be deep neural networks, which work in the regime of high expressiveness and efficient computation (assuming specialized hardware)
- How exactly do we design these networks?
 - Any setting of θ must define a valid probability distribution over x :

$$\text{for all } \theta, \quad \sum_{\mathbf{x}} p_\theta(\mathbf{x}) = 1 \quad \text{and} \quad p_\theta(\mathbf{x}) \geq 0 \quad \text{for all } \mathbf{x}$$

- $\log p_\theta(x)$ should be easy to evaluate and differentiate with respect to θ
- This can be tricky to set up!

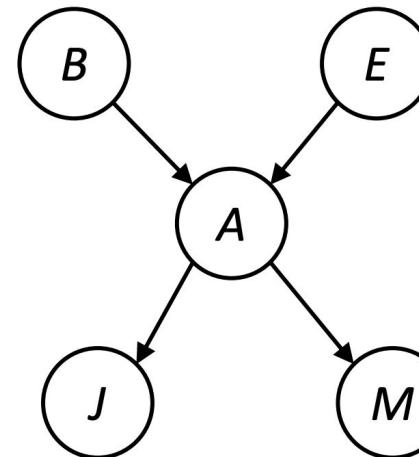
Bayes nets and neural nets

Main idea: place a **Bayes net** structure (a directed acyclic graph) over the variables in the data, and model the conditional distributions with neural networks.

Reduces the problem to designing coin variables. We know how to do this: the variable as input, and outputs the distribution

B	P(B)
+b	0.001
-b	0.999

A	J	P(J A)
+a	+j	0.9
+a	-j	0.1
-a	+j	0.05
-a	-j	0.95



E	P(E)
+e	0.002
-e	0.998

A	M	P(M A)
+a	+m	0.7
+a	-m	0.3
-a	+m	0.01
-a	-m	0.99

Autoregressive models

- First, given a Bayes net structure, setting the conditional distributions to neural networks will yield a tractable log likelihood and gradient. Great for maximum likelihood training!

$$\log p_{\theta}(\mathbf{x}) = \sum_{i=1}^d \log p_{\theta}(x_i \mid \text{parents}(x_i))$$

- But is it expressive enough? Yes, assuming a fully expressive Bayes net structure: any joint distribution can be written as a product of conditionals

$$\log p(\mathbf{x}) = \sum_{i=1}^d \log p(x_i \mid \mathbf{x}_{1:i-1})$$

- This is called an **autoregressive model**. So, an expressive Bayes net structure with neural network conditional distributions yields an expressive model for $p(\mathbf{x})$ with tractable maximum likelihood training.

A toy autoregressive model

Two variables: x_1, x_2

Model: $p(x_1, x_2) = p(x_1) p(x_2 | x_1)$

- $p(x_1)$ is a histogram
- $p(x_2 | x_1)$ is a multilayer perceptron
 - Input is x_1
 - Output is a distribution over x_2 (logits, followed by softmax)

One function approximator per conditional

Does this extend to high dimensions?

- Somewhat. For d -dimensional data, $O(d)$ parameters
 - Much better than $O(\exp(d))$ in tabular case
 - What about text generation where d can be arbitrarily large?
- Limited generalization
 - No information sharing among different conditionals

Solution: share parameters among conditional distributions. Two approaches:

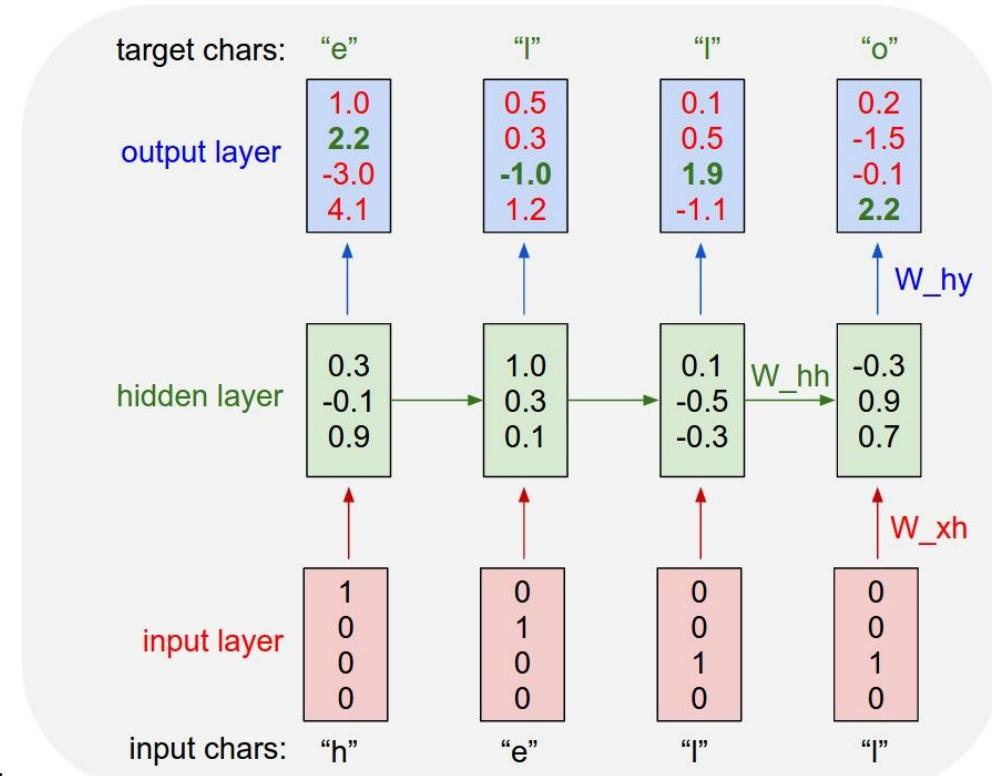
- Recurrent neural networks
- Masking

RNN autoregressive models - char-rnn

$$\log p(\mathbf{x}) = \sum_{i=1}^d \log p(x_i | \mathbf{x}_{1:i-1})$$

Sequence of characters

Character at i^{th} position



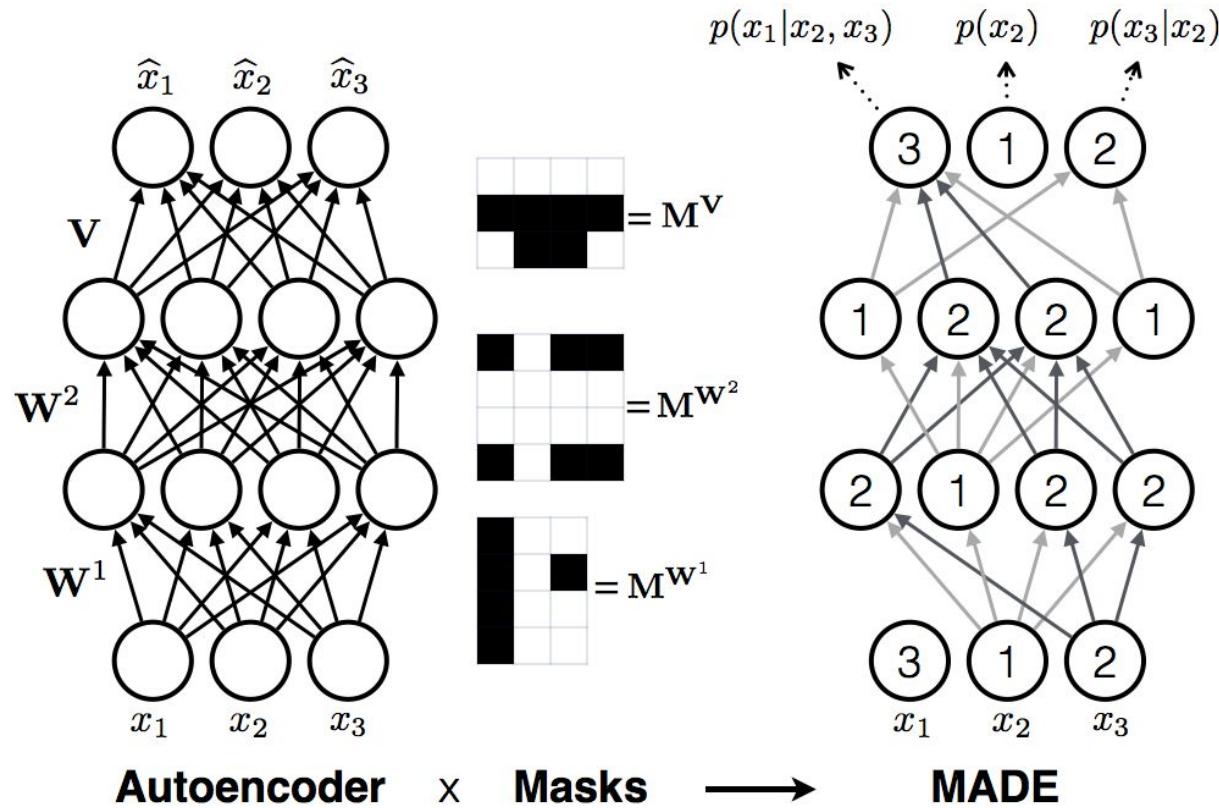
[Karpathy, 2015]

Masking-based autoregressive models

Second major branch of neural AR models

- Key property: parallelized computation of all conditionals
- Masked MLP (MADE)
- Masked convolutions & self-attention
 - Also share parameters across time

Masked Autoencoder for Distribution Estimation (MADE)



Masked Autoencoder for Distribution Estimation (MADE)

```
# param: normal FC weight
# x: layer input
# y: autoregressive activations
mask = get_linear_ar_mask(in_size, out_size)
# create mask of pattern
# array([[0., 1., 1.],
#        [0., 0., 1.],
#        [0., 0., 0.]], dtype=float32)
y = tf.matmul(x, param * mask)
```

MADE results

Table 6. Negative log-likelihood test results of different models on the binarized MNIST dataset.

Model	$-\log p$	
RBM (500 h, 25 CD steps)	≈ 86.34	Intractable
DBM 2hl	≈ 84.62	
DBN 2hl	≈ 84.55	
DARN $n_h=500$	≈ 84.71	
DARN $n_h=500$, adaNoise	≈ 84.13	
MoBernoullis K=10	168.95	Tractable
MoBernoullis K=500	137.64	
NADE 1hl (fixed order)	88.33	
EoNADE 1hl (128 orderings)	87.71	
EoNADE 2hl (128 orderings)	85.10	
MADE 1hl (1 mask)	88.40	Tractable
MADE 2hl (1 mask)	89.59	
MADE 1hl (32 masks)	88.04	
MADE 2hl (32 masks)	86.64	

MADE results

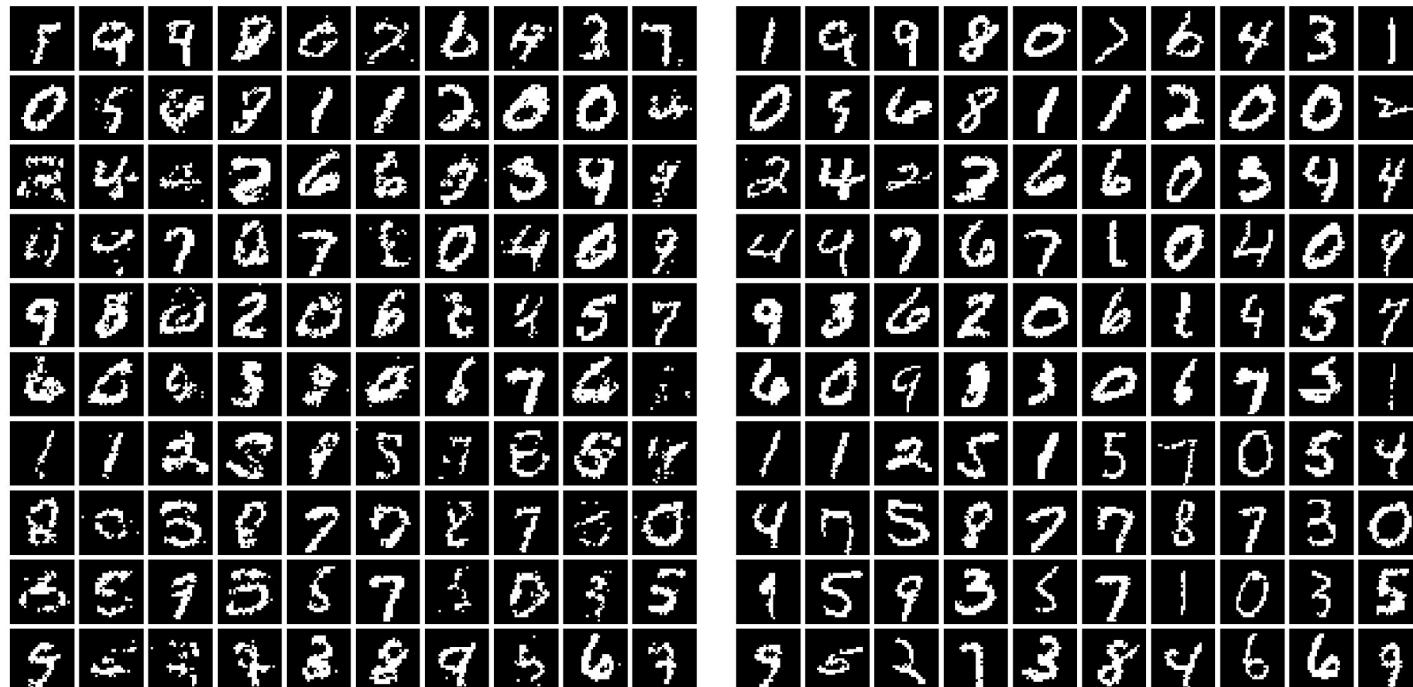
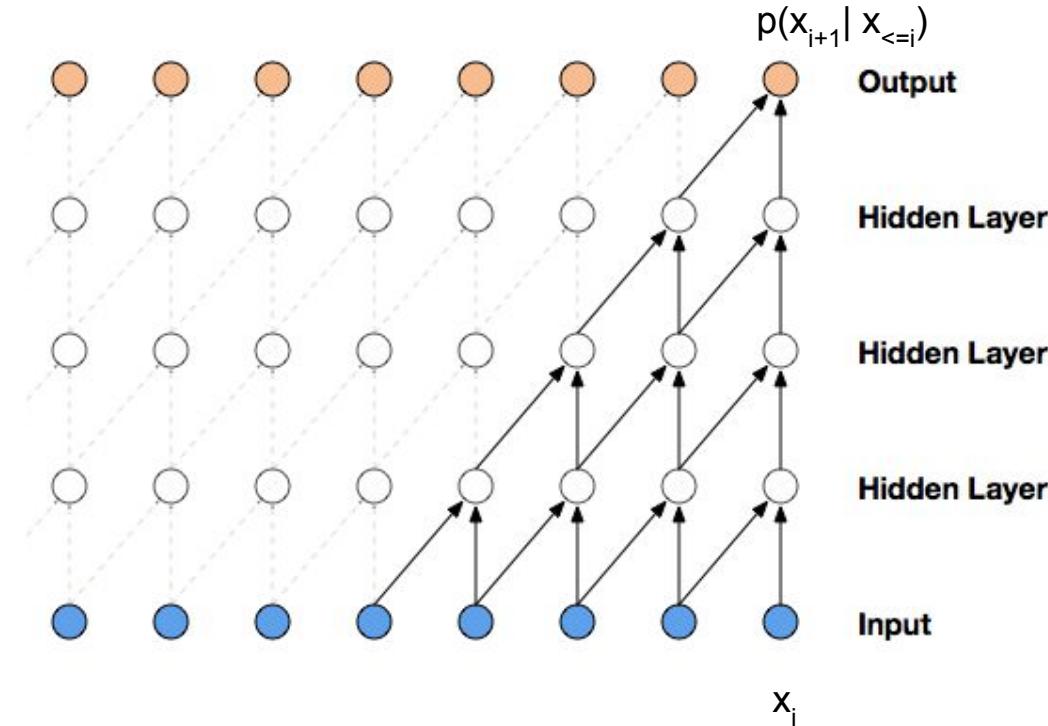


Figure 3. Left: Samples from a 2 hidden layer MADE. Right: Nearest neighbour in binarized MNIST.

Masked Temporal (1D) Convolution

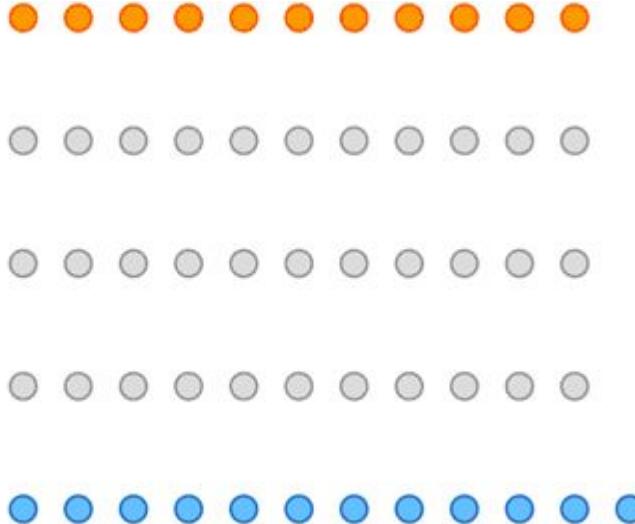


- Easy to implement, masking part of the conv kernel
- Constant parameter count for variable-length distribution!
- Efficient to compute, convolution has hyper-optimized implementations on all hardware

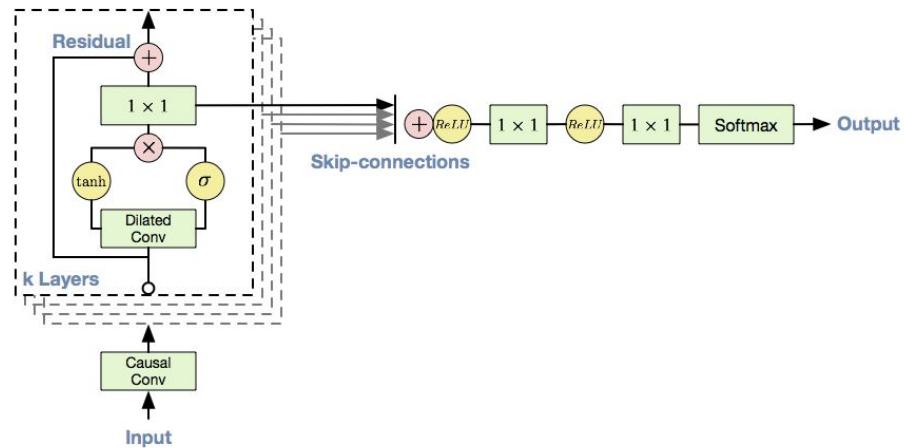
However

- Limited receptive field, linear in number of layers

WaveNet



- Improved receptive field: dilated convolution, with exponential dilation
- Better expressivity: Gated Residual blocks, Skip connections

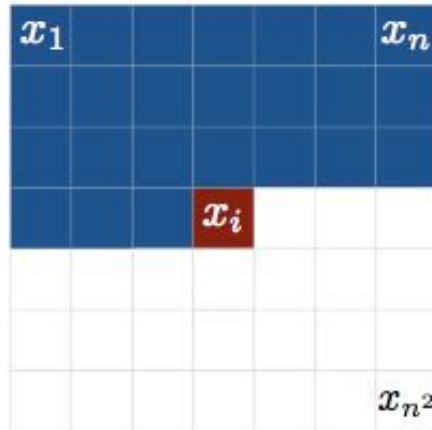


Masked Temporal (1D) Convolution

```
# More efficient implementation possible by
# padding instead of masking kernels
# k: size of kernel
# kernel: convolution weights
padded_x = tf.pad(x, [
    (0, 0), (k - 1, 0),
    (0, 0), (0, 0)
])
y = tf.nn.conv2d(padded_x, kernel, padding='VALID')
```

Masked Spatial (2D) Convolution - PixelCNN

- Images can be flatten into 1D vectors, but they are fundamentally 2D
- We can use a masked variant of ConvNet to exploit this knowledge
- First, we impose an autoregressive ordering on 2D images:

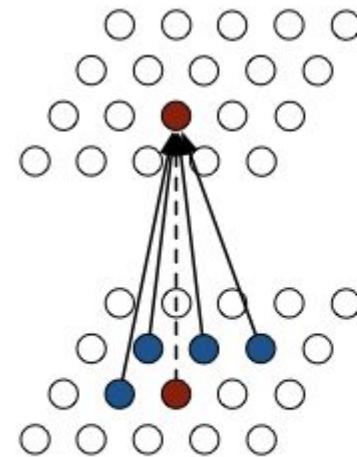


This is called raster scan ordering.
(Different orderings are possible,
more on this later)

PixelCNN

- Design question: how to design a masking method to obey that ordering?
- One possibility: PixelCNN (2016)

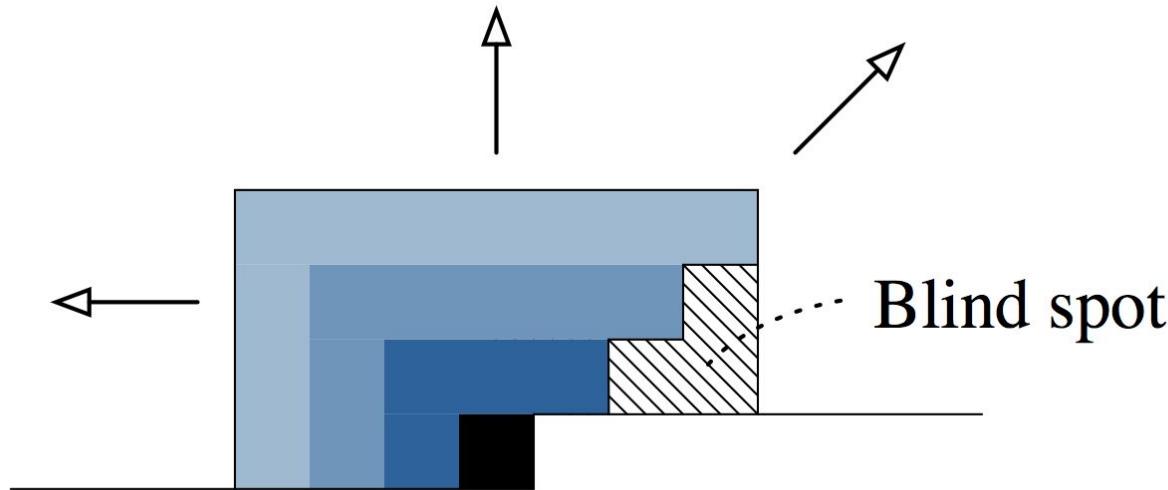
1	1	1
1	0	0
0	0	0



PixelCNN

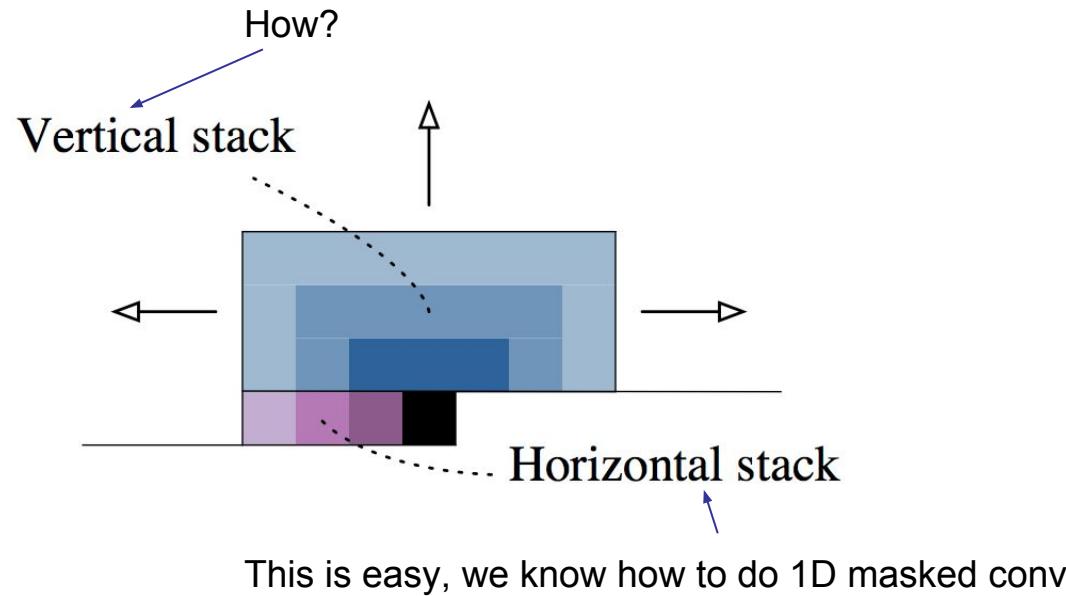
PixelCNN

- PixelCNN-style masking has one problem: blind spot in receptive field



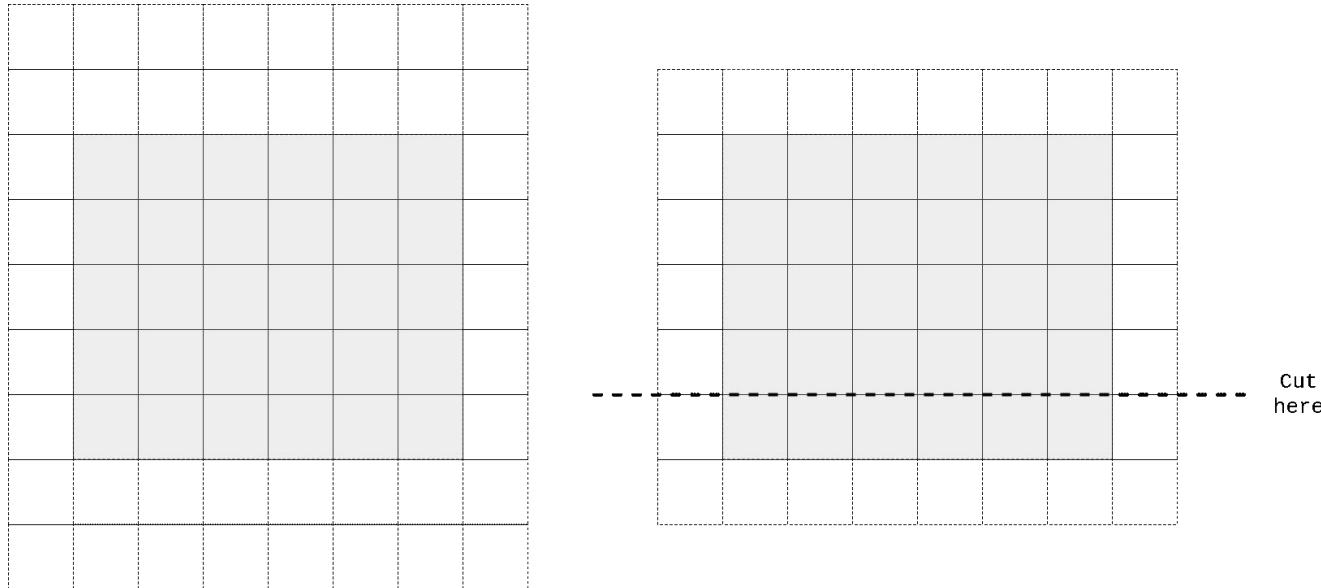
Gated PixelCNN

- Gated PixelCNN (2016) introduced a fix by combining two streams of convolutions



Gated PixelCNN

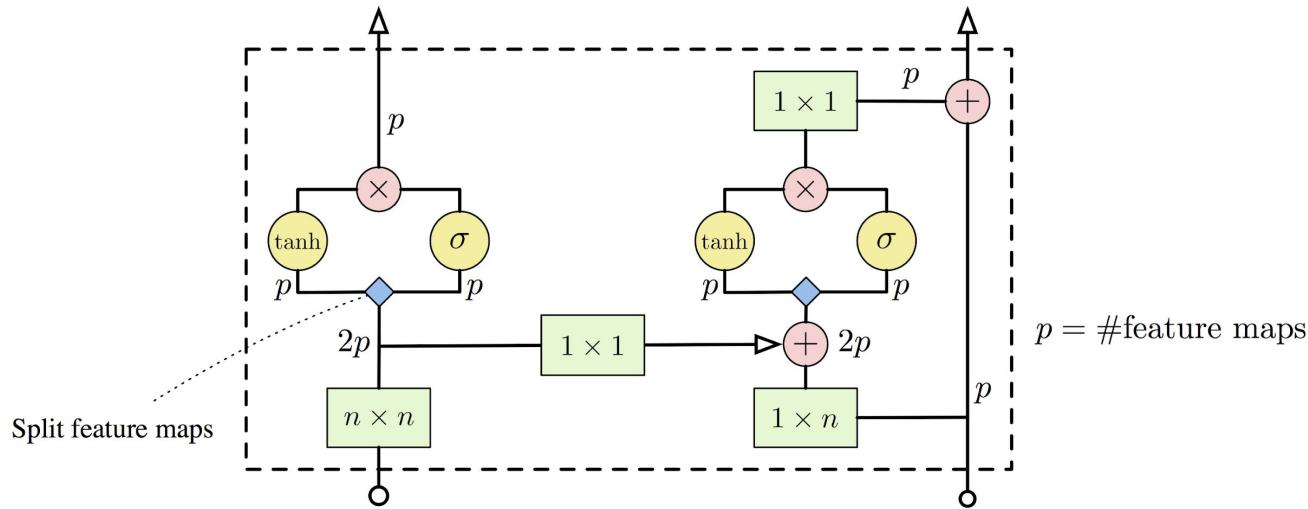
- Vertical stack: through padding, activations at i th row only depend on input before i th row



Gated PixelCNN

- Improved ConvNet architecture: Gated ResNet Block

$$\mathbf{y} = \tanh(W_{k,f} * \mathbf{x}) \odot \sigma(W_{k,g} * \mathbf{x})$$



Gated PixelCNN

- Better receptive field + more expressive architecture = better performance

Model	NLL Test (Train)
Uniform Distribution: [30]	8.00
Multivariate Gaussian: [30]	4.70
NICE: [4]	4.48
Deep Diffusion: [24]	4.20
DRAW: [9]	4.13
Deep GMMs: [31, 29]	4.00
Conv DRAW: [8]	3.58 (3.57)
RIDE: [26, 30]	3.47
PixelCNN: [30]	3.14 (3.08)
PixelRNN: [30]	3.00 (2.93)
Gated PixelCNN:	3.03 (2.90)

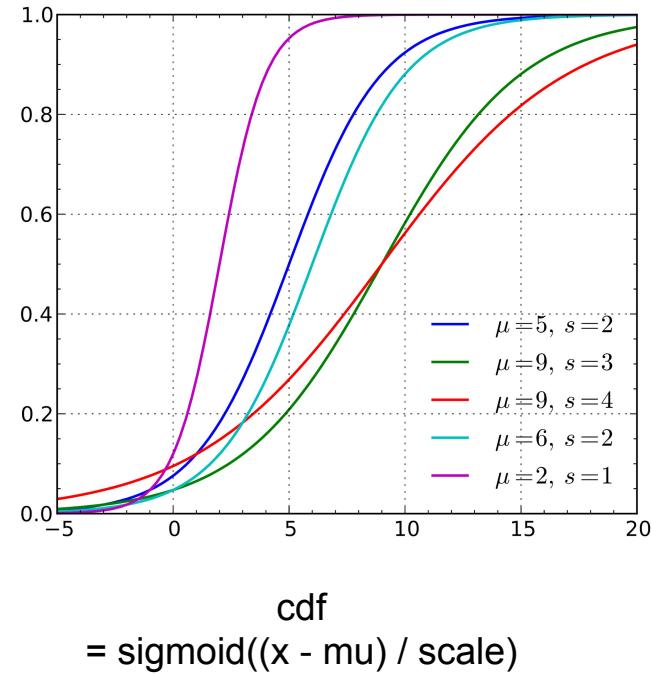
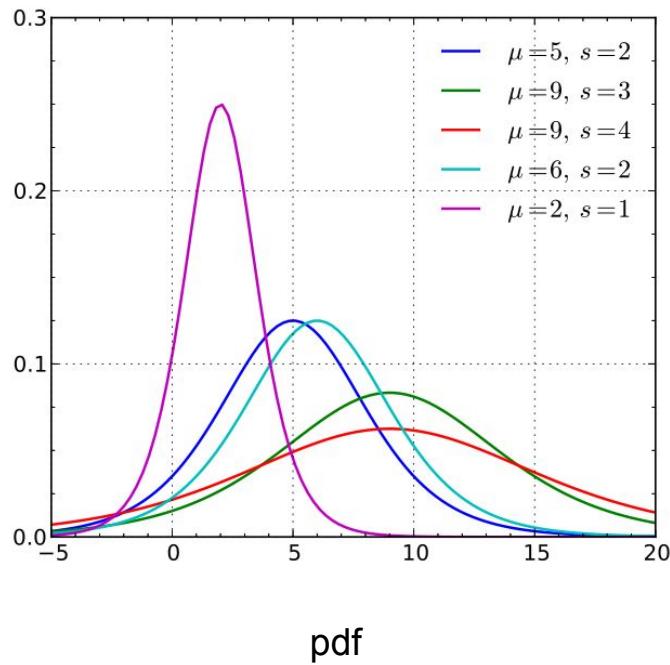
PixelCNN++

- Moving away from softmax: we know nearby pixel values are likely to co-occur!

$$\nu \sim \sum_{i=1}^K \pi_i \text{logistic}(\mu_i, s_i)$$

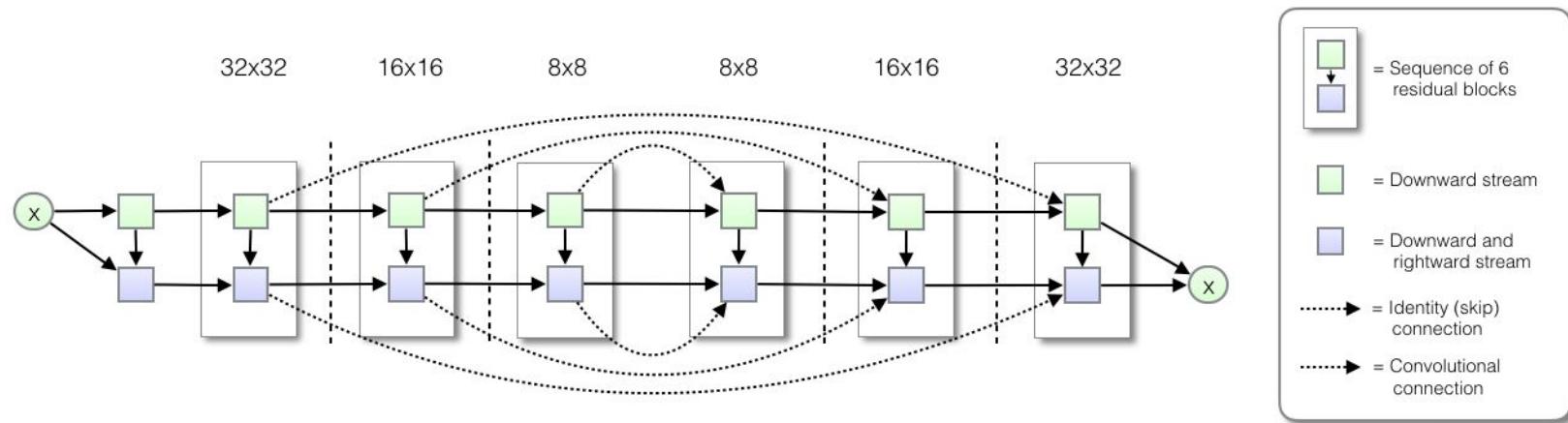
$$P(x|\pi, \mu, s) = \sum_{i=1}^K \pi_i [\sigma((x + 0.5 - \mu_i)/s_i) - \sigma((x - 0.5 - \mu_i)/s_i)] ,$$

Recap: Logistic distribution



PixelCNN++

- Capture long dependencies efficiently by downsampling



PixelCNN++

Model	Bits per sub-pixel
Deep Diffusion (Sohl-Dickstein et al., 2015)	5.40
NICE (Dinh et al., 2014)	4.48
DRAW (Gregor et al., 2015)	4.13
Deep GMMs (van den Oord & Dambre, 2015)	4.00
Conv DRAW (Gregor et al., 2016)	3.58
Real NVP (Dinh et al., 2016)	3.49
PixelCNN (van den Oord et al., 2016b)	3.14
VAE with IAF (Kingma et al., 2016)	3.11
Gated PixelCNN (van den Oord et al., 2016c)	3.03
PixelRNN (van den Oord et al., 2016b)	3.00
PixelCNN++	2.92

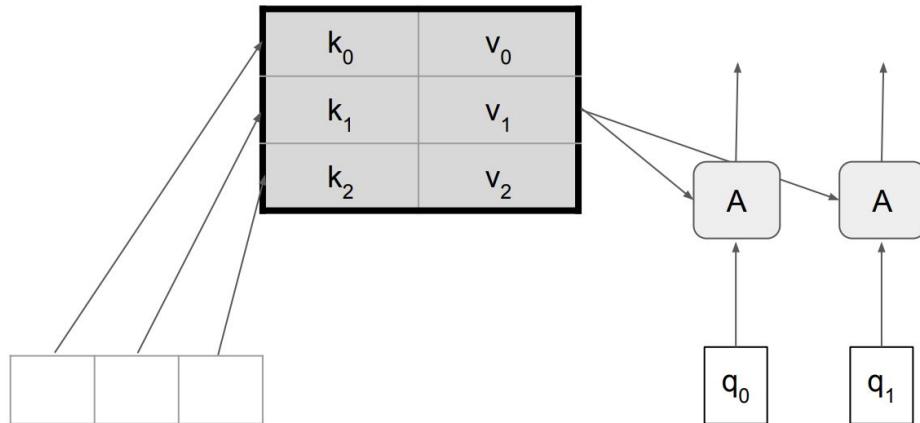
Masked Attention

- A recurring problem for convolution: limited receptive field -> hard to capture long-range dependencies
- (Self-)Attention: an alternative that has
 - unlimited receptive field!!
 - also $O(1)$ parameter scaling w.r.t. data dimension
 - parallelized computation (versus RNN)

Attention

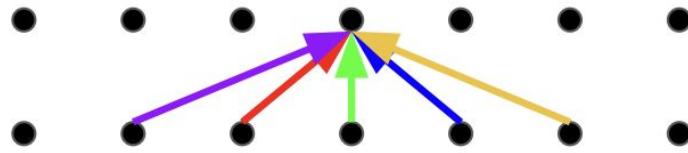
Dot-Product Attention

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i}}{\sum_j e^{q \cdot k_j}} v_i$$

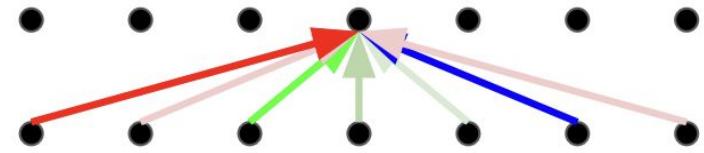


Self-attention when q_i also generated from x

Self-Attention



Convolution

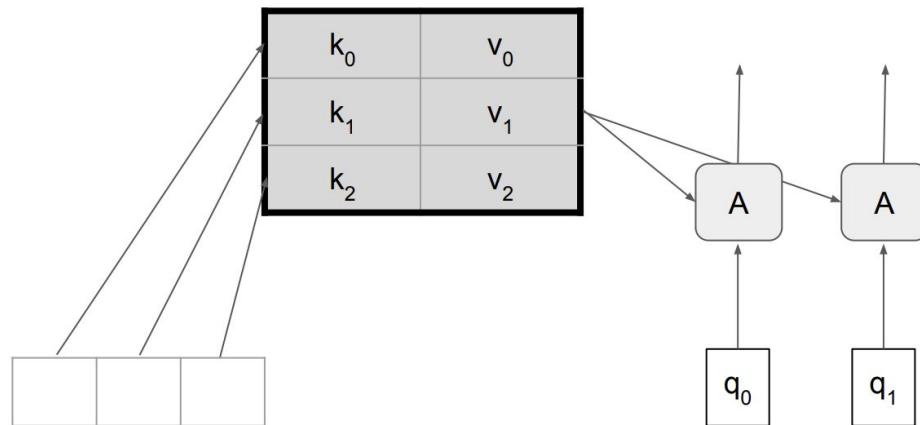


Self-attention

Masked Attention

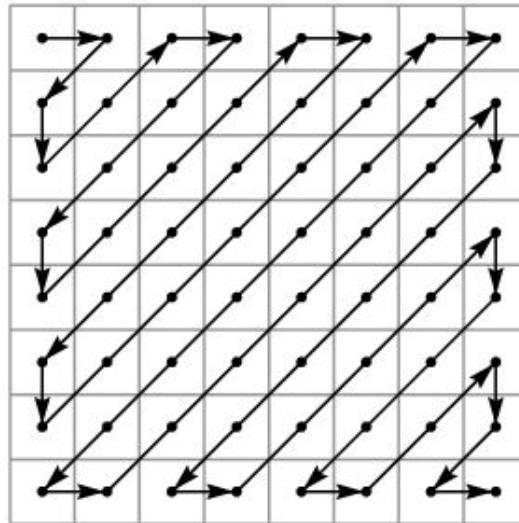
Dot-Product Attention

$$A(q, K, V) = \sum_i \frac{e^{q \cdot k_i - \text{masked}(k_i, q) * 10^{10}}}{\sum_j e^{q \cdot k_j - \text{masked}(k_j, q) * 10^{10}}} v_i$$



Masked Attention

- Much more flexible than masked convolution. We can design any autoregressive ordering we want
- An example:

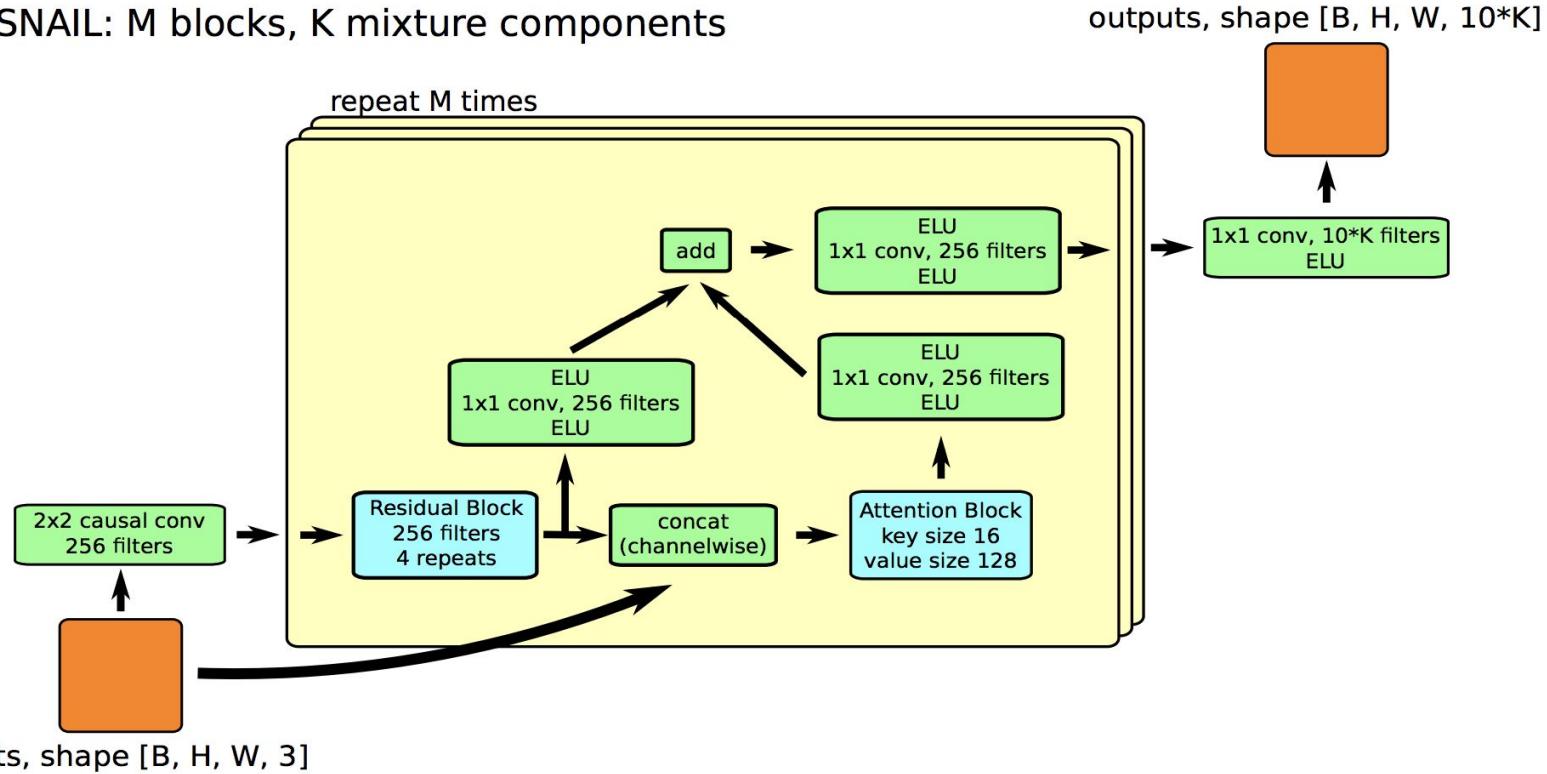


Zigzag ordering

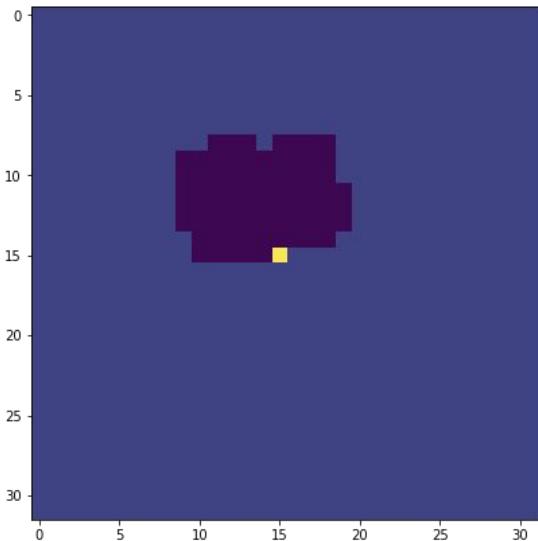
- How to implement with masked conv?
- Trivial to do with masked attention!

Masked Attention + Convolution

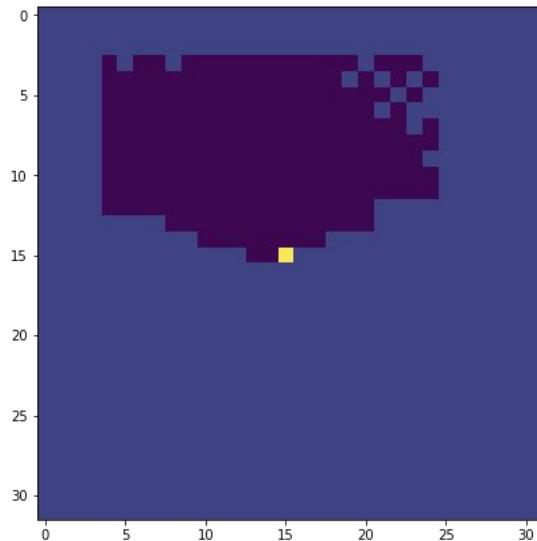
PixelSNAIL: M blocks, K mixture components



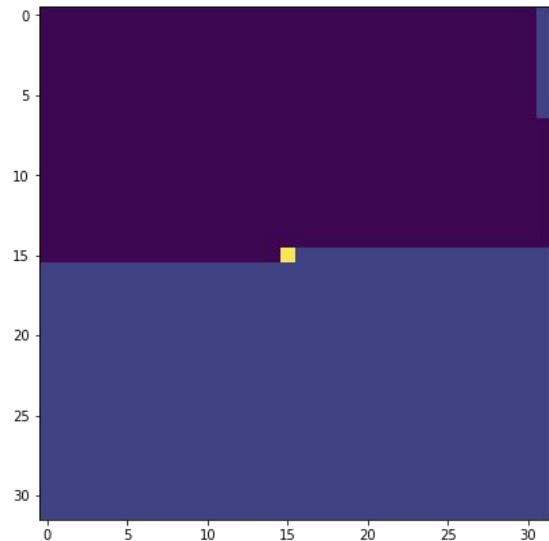
Masked Attention + Convolution



Gated PixelCNN



PixelCNN++



PixelSNAIL

Masked Attention + Convolution

Method	CIFAR-10
Conv DRAW (Gregor et al., 2016)	3.5
Real NVP (Dinh et al., 2016)	3.49
VAE with IAF (Kingma et al., 2016)	3.11
PixelRNN (Oord et al., 2016b)	3.00
Gated PixelCNN (van den Oord et al., 2016b)	3.03
Image Transformer (Anonymous, 2018)	2.98
PixelCNN++ (Salimans et al., 2017)	2.92
Block Sparse PixelCNN++ (OpenAI, 2017)	2.90
PixelSNAIL (ours)	2.85

Neural autoregressive models: the good

Best in class modelling performance:

- expressivity - autoregressive factorization is general
- generalization - meaningful parameter sharing has good inductive bias

-> State of the art models on multiple datasets, modalities

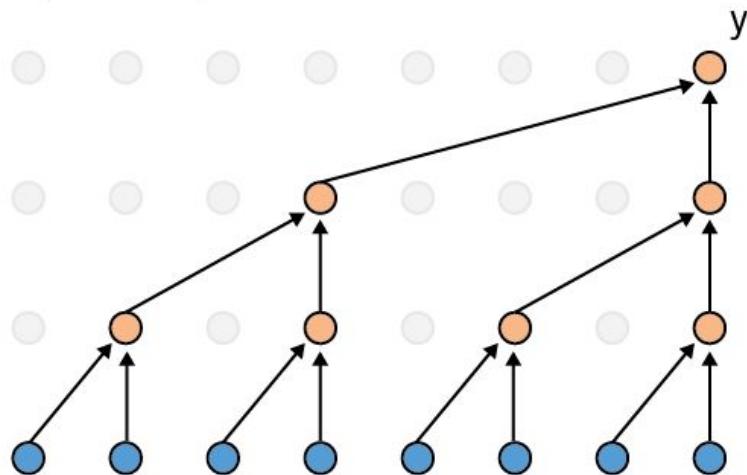
Masked autoregressive models: the bad

- Sampling each pixel = 1 forward pass!
- 11 minutes to generate 16 32-by-32 images on a Tesla K40 GPU

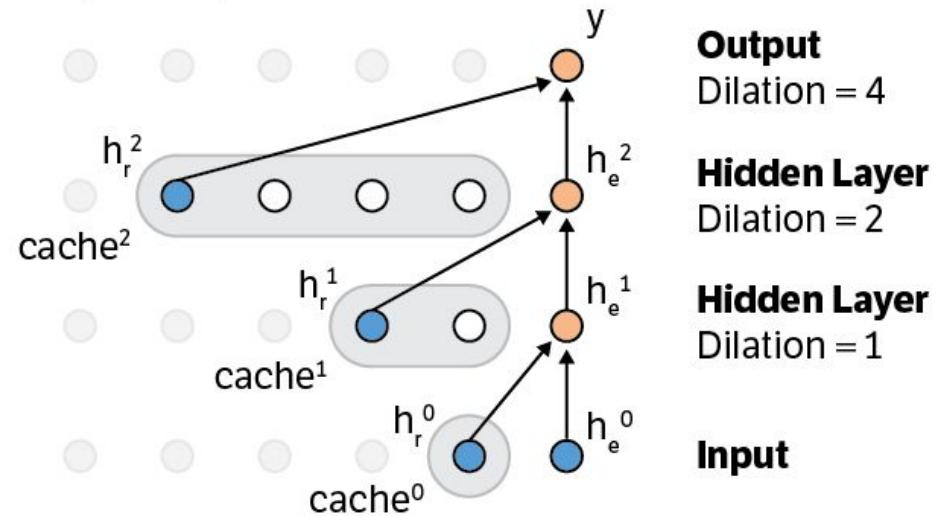
Speedup by caching activations

<https://github.com/PrajitR/fast-pixel-cnn>

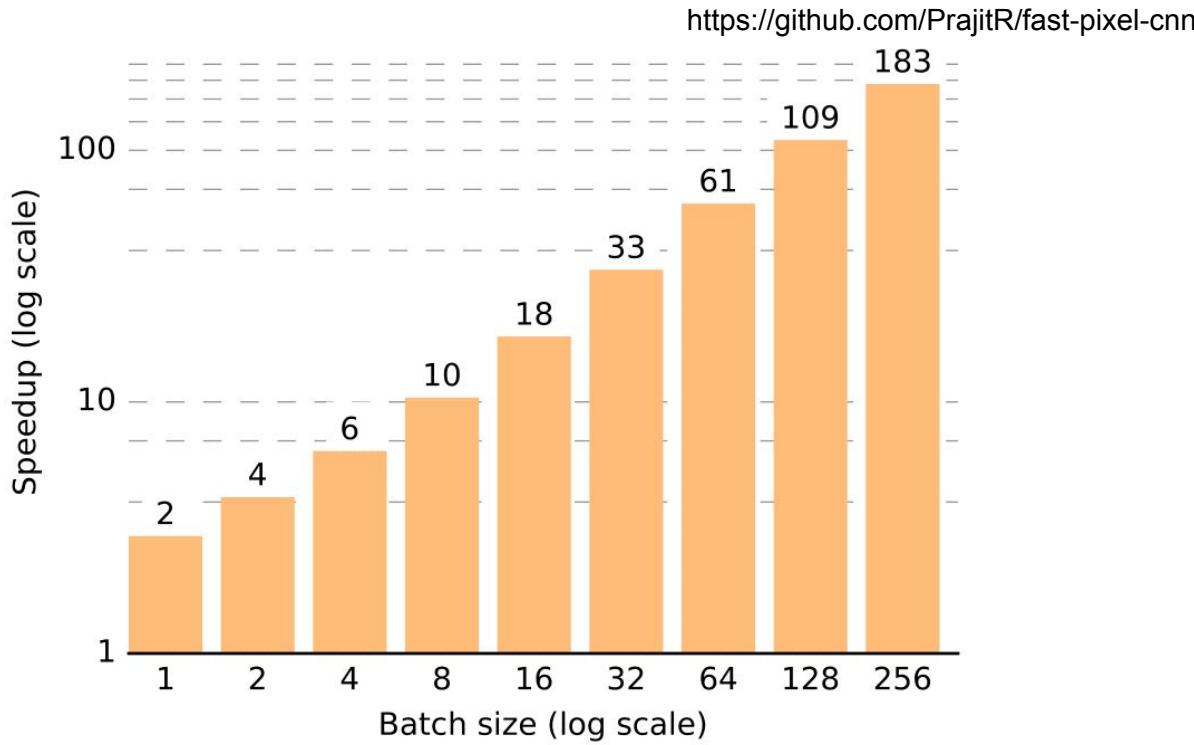
A) Naive Implementation



B) Our Implementation

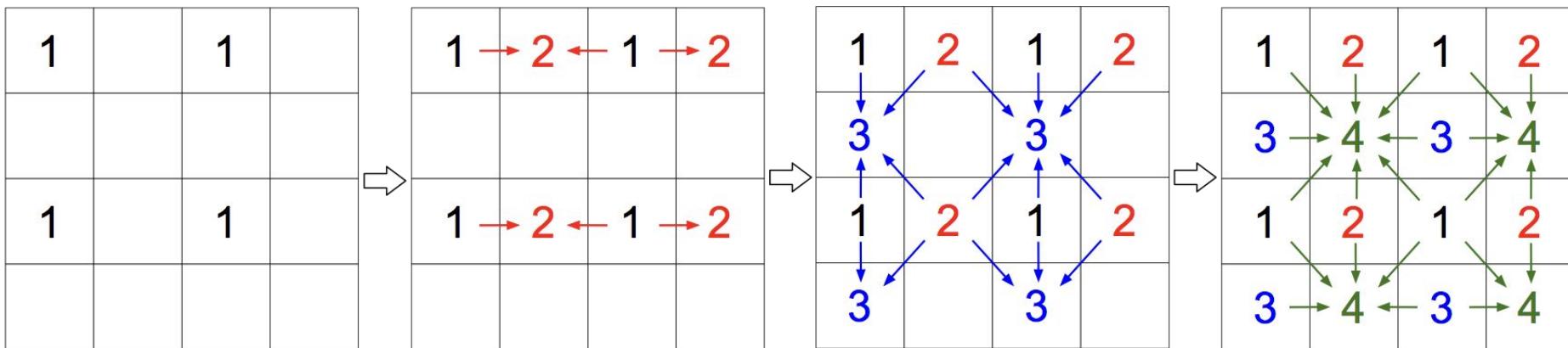


Speedup by caching activations



Speedup by breaking autoregressive pattern

- $O(d) \rightarrow O(\log(d))$ by parallelizing within groups {2, 3, 4}
- Cannot capture dependencies within each group: this is a fine assumption if all pixels in one group are conditionally independent
 - Most often they are not, then you trade expressivity for sampling speed



Multiscale PixelCNN

Model	scale	time	speedup
$O(N)$ PixelCNN	32	120.0	1.0x
$O(\log N)$ PixelCNN	32	1.17	102x
$O(\log N)$ PixelCNN, in-graph	32	1.14	105x

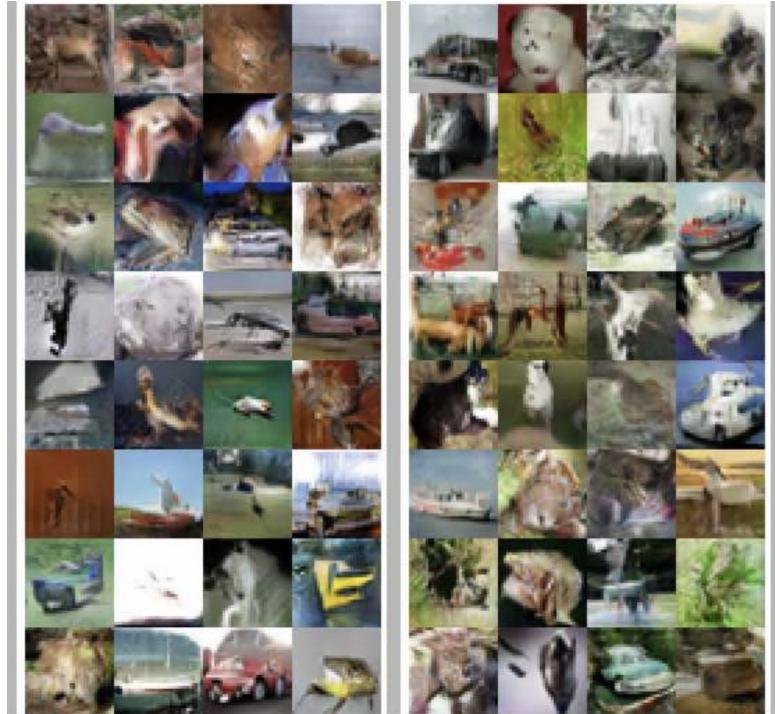
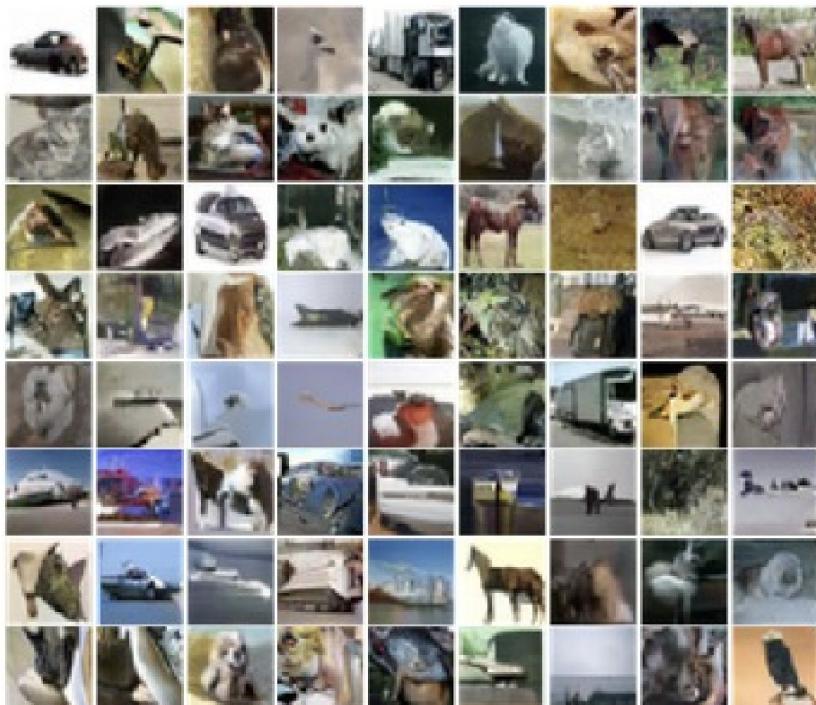
Improved sampling speed

Model	32	64	128
PixelRNN	3.86 (3.83)	3.64(3.57)	-
PixelCNN	3.83 (3.77)	3.57(3.48)	-
Real NVP	4.28(4.26)	3.98(3.75)	-
Conv. DRAW	4.40(4.35)	4.10(4.04)	-
Ours	3.95(3.92)	3.70(3.67)	3.55(3.42)

Table 3. ImageNet negative log-likelihood in bits per sub-pixel at 32×32 , 64×64 and 128×128 resolution.

More limited modelling capacity

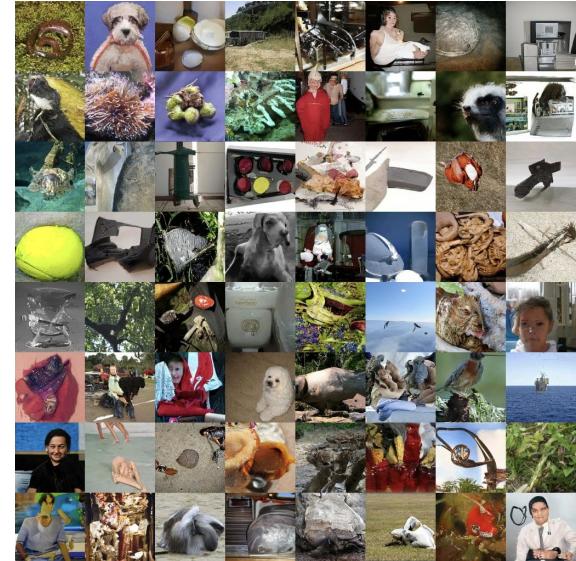
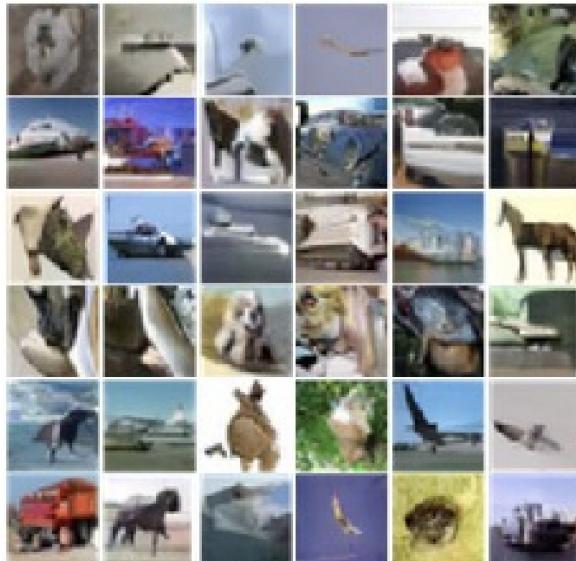
Sample Quality



Which set of samples are generated by a GAN versus an AR model?

AR models can have good samples

- Good samples can be achieved by selective bits conditioning
 - Grayscale PixelCNN
 - Subscale Pixel Network



Bibliography

char-rnn: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

MADE: Germain, Mathieu, et al. "Made: Masked autoencoder for distribution estimation." *International Conference on Machine Learning*. 2015.

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

PixelCNN: Oord, Aaron van den, Nal Kalchbrenner, and Koray Kavukcuoglu. "Pixel recurrent neural networks." *arXiv preprint arXiv:1601.06759* (2016).

Gated PixelCNN: Van den Oord, Aaron, et al. "Conditional image generation with pixelcnn decoders." *Advances in Neural Information Processing Systems*. 2016.

PixelCNN++: Salimans, Tim, et al. "Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications." *arXiv preprint arXiv:1701.05517* (2017)

Self-attention: Vaswani, Ashish, et al. "Attention is all you need." *Advances in Neural Information Processing Systems*. 2017.

PixelSNAIL: Chen, Xi, et al. "Pixelsnail: An improved autoregressive generative model." *arXiv preprint arXiv:1712.09763* (2017)

Fast PixelCNN++: Ramachandran, Prajit, et al. "Fast generation for convolutional autoregressive models." *arXiv preprint arXiv:1704.06001*(2017).

Multiscale PixelCNN: Reed, Scott, et al. "Parallel multiscale autoregressive density estimation." *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017.

Grayscale PixelCNN: Kolesnikov, Alexander, and Christoph H. Lampert. "PixelCNN models with auxiliary variables for natural image modeling." *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017.

Subscale Pixel Network: Menick, Jacob, and Nal Kalchbrenner. "Generating High Fidelity Images with Subscale Pixel Networks and Multidimensional Upscaling." *arXiv preprint arXiv:1812.01608*(2018)