Advanced Course on Deep Generative Models

Lecture 4: Autoregressive models

Adapted from Volodymyr Kuleshov

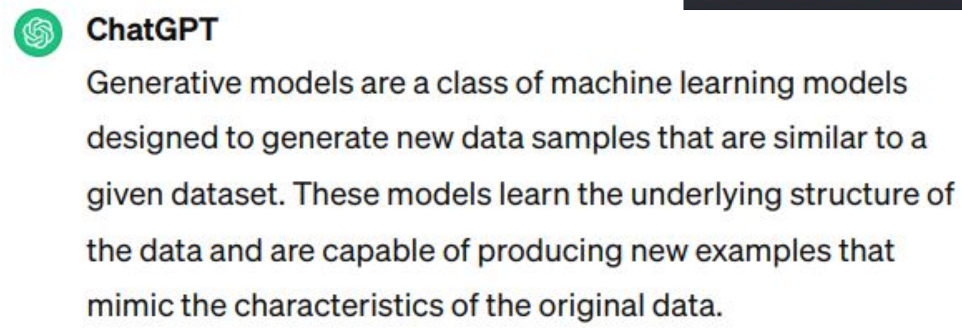
Dan Rosenbaum, CS Haifa

Today

- Recap
- Autoregressive models with logistic regression
- Deep learning
- NADE, MADE, RNNs
- PixelCNN

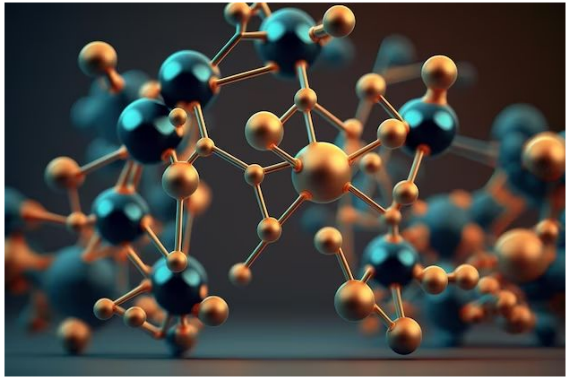
What are generative models?

- High dimensional output
- Probabilistic



ChatGPT

Generative models are a class of machine learning models designed to generate new data samples that are similar to a given dataset. These models learn the underlying structure of the data and are capable of producing new examples that mimic the characteristics of the original data.

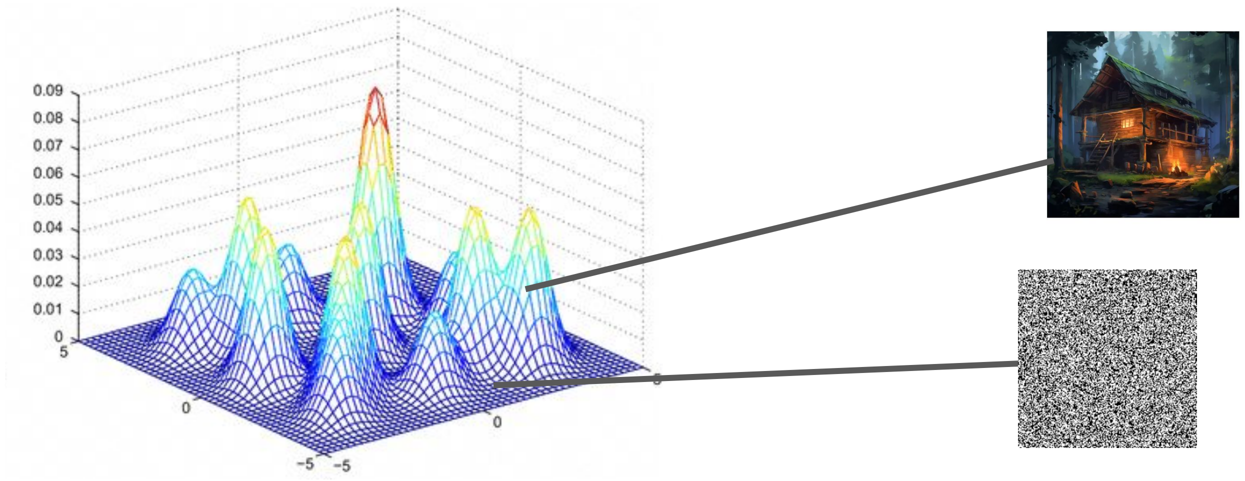


What can we do with generative models?

- Sample (generate data)
- Density estimation (probabilistic reasoning, decision making)
- Representation learning (downstream tasks)

Learning a generative model

- Data is generated by an unknown underlying distribution $\mathbf{p}_{\text{data}}$
- We are looking for the parameters $\boldsymbol{\theta}$ such that $\mathbf{p}_{\boldsymbol{\theta}}$ is close to $\mathbf{p}_{\text{data}}$



Components for training a generative model

1. Data - representative of the space
2. Model (e.g. Gaussian, mixture of Gaussians, Latent variable model)
3. Objective (e.g. maximum likelihood, score matching)
4. Optimization (e.g. Variational inference, MCMC)

Challenge: Solve the curse of dimensionality.

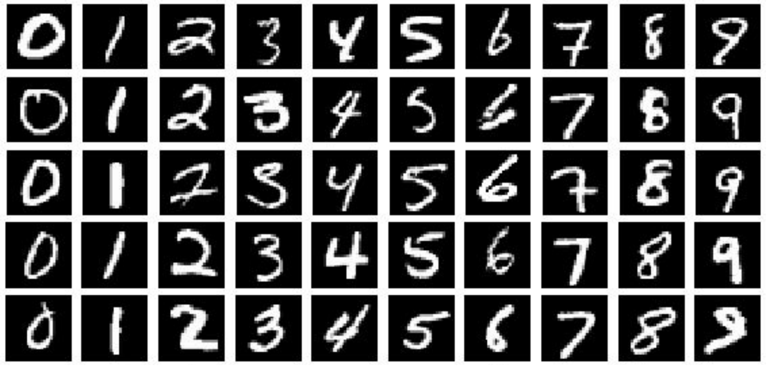
Data

Two standard ways to represent images:

1. Discrete. 8-bit integers {0…255}

2. Continuous. Values in [0, 1]

RGB images are 3 channels per pixel



Models

- Discrete:
    - Bernoulli
    - Categorical

- Continuous:
    - Gaussian
    - GMM

So far we have seen simple parameterization of these models

Today we will start looking at a parameterization based on deep NNs.

Structure through conditional independence

- Using Chain Rule

$$p(x_1, \ldots, x_n) = p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1, x_2) \cdots p(x_n \mid x_1, \cdots, x_{n-1})$$

- How many parameters? $1 + 2 + \cdots + 2^{n-1} = 2^n - 1$
  - $p(x_1)$ requires 1 parameter
  - $p(x_2 \mid x_1 = 0)$ requires 1 parameter, $p(x_2 \mid x_1 = 1)$ requires 1 parameter Total 2 parameters.
  - $\cdots$
- $2^n - 1$ is still exponential, chain rule does not buy us anything.

Options to reduce number of parameters

Conditional independence given **observed** variables:	Conditional independence given **latent** variables:	Constrained parameterization:
$p(x_1, \ldots, x_n) = \Pi_i\, p(x_i \mid x_{[i-k:i-1]})$	$p(x_1, \ldots, x_n) = \Pi_i\, p(x_i \mid z)$	$p_\theta(x_i \mid z)$
For Discrete data: $d^k \cdot (d-1) \cdot n$ parameters	For Discrete data: $\lvert h \rvert \cdot (d-1) \cdot n$ parameters	$\lvert\theta\rvert$ parameters

Objective - training a model

Given a model $\mathbf{p}_{\boldsymbol{\theta}}$ with unknown parameters $\boldsymbol{\theta}$, find the values of $\boldsymbol{\theta}$ that make it as close as possible to $\mathbf{p}_{\mathbf{data}}$

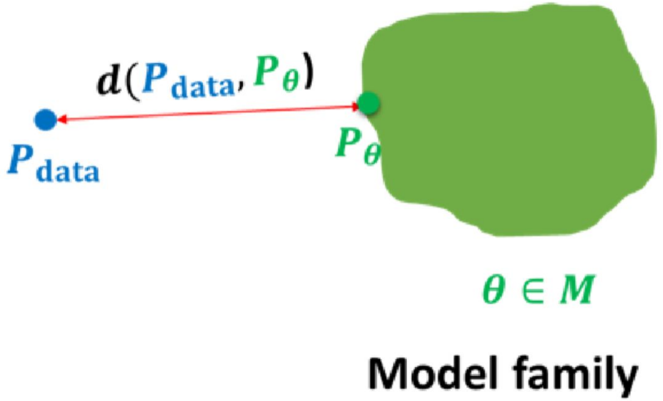
How can we do it?

Standard approach: Maximum likelihood.

Maximum Likelihood

We want to minimize some distance between $\mathbf{p}_\theta$ and $\mathbf{p}_{\text{data}}$

$x_i \sim P_{\text{data}}$
$i = 1, 2, \ldots, n$



How do we measure the distance between distributions?

credit: Volodymyr Kuleshov

Maximum Likelihood

- KL divergence:

$$\begin{aligned} D_{KL}(p_{\text{data}}, p_\theta) &= \int p_{\text{data}}(x) \log \frac{p_{\text{data}}(x)}{p_\theta(x)} dx \\ &= -\int p_{\text{data}}(x) \log p_\theta(x) dx + \text{const.} \\ &\approx -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i) + \text{const.}, \quad x_i \sim p_{\text{data}} \end{aligned}$$

- For a discrete representation: # of bits required to communicate image **cross-entropy**

Constrained Parameterization

If we don't use a full parameterization, we can use less parameters without making independence assumptions.

Example, logistic regression:

$$f(x) = \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)}$$

Running example: MNIST

- Suppose we are given a dataset N of handwritten digits (binarized MNIST, 255 MNIST)
- Each image has $n = 28 \times 28 = 784$ pixels.
  Binary case: each pixel can either be black (0) or white (1).
- Our Goal: Learn a probability distribution $\boldsymbol{p(x) = p(x_1, \cdots, x_{784})}$ over $\boldsymbol{x \in \{0, 1\}^{784}}$ such that when $\boldsymbol{x \sim p(x)}$, $\boldsymbol{x}$ looks like a digit

An autoregressive model from logistic regression

- Always true:

$$p(x) = p(x_1, \cdots, x_{784}) = p(x_1)p(x_2|x_1) \cdots p(x_{784}|x_{1:783})$$

- Too complex to store in tabular form

- Parameterize conditional using logistic regression:

$$p(x) = p(x_1, \cdots, x_{784}) = p_{logit}(x_1;\theta_1)p_{logit}(x_2|x_1;\theta_2) \cdots p_{logit}(x_{784}| x_{1:783};\theta_{783})$$

where $p_{logit}(x_i = 1 \,|\, x_{<i};\theta_i) = \sigma(\theta_i^T [1, x_{<i}])$

# of parameters: $1 + 2 + .. + n \approx n^2/2$

An autoregressive model from logistic regression

Each term

$$p_{logit}(x_i = 1 \mid x_{<i}; \theta_i) = \sigma(\theta_i^T [1, x_{<i}])$$

Is a Bernoulli distribution with parameter $\sigma(\theta_i^T [1, x_{<i}])$

Interpretations:

1. The probability that pixel i is on
2. A prediction of the next pixel $\mathbf{x}_i$ given previous pixels $\mathbf{x}_{<i}$

Fully Visible Sigmoid Belief Network (FVSBN)

How do we evaluate $\boldsymbol{p(x) = p(x_1, \cdots, x_{784})}$?

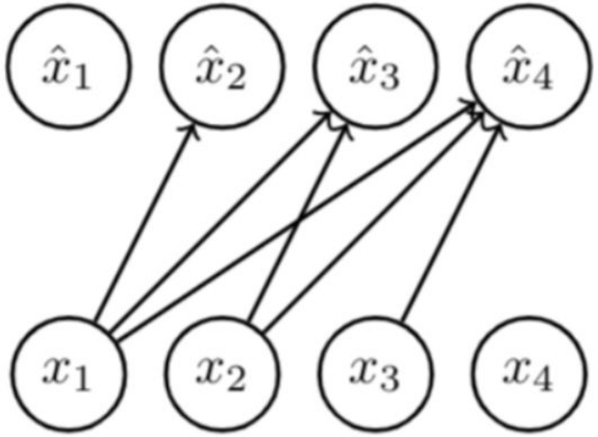
1. First compute the output (Bernoulli parameter) for each factor given $\boldsymbol{x_{<i}}$

2. Then evaluate the probability of each factor given the observed $\mathbf{x_i}$

Can be done in parallel for all $\boldsymbol{x_i}$

How do we train the model?

⇒ maximum likelihood by gradient descent.



FVSBN

Fully Visible Sigmoid Belief Network (FVSBN)

How do we sample from $p(x) = p(x_1, \cdots, x_{784})$?

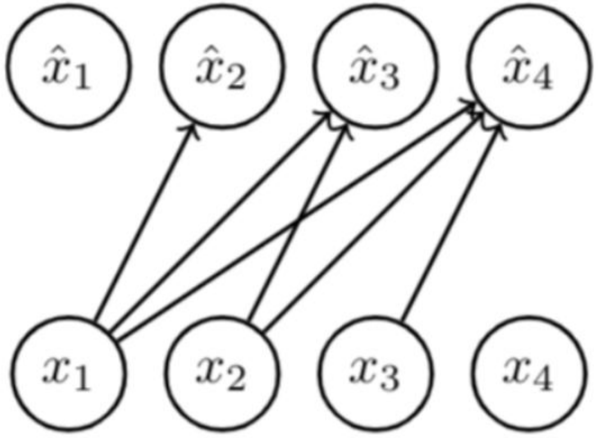
1. Sample $x_1 \sim p(x_1)$
2. Sample $x_2 \sim p(x_2 | x_1)$
3. Sample $x_3 \sim p(x_3 | x_1, x_2)$

.

.

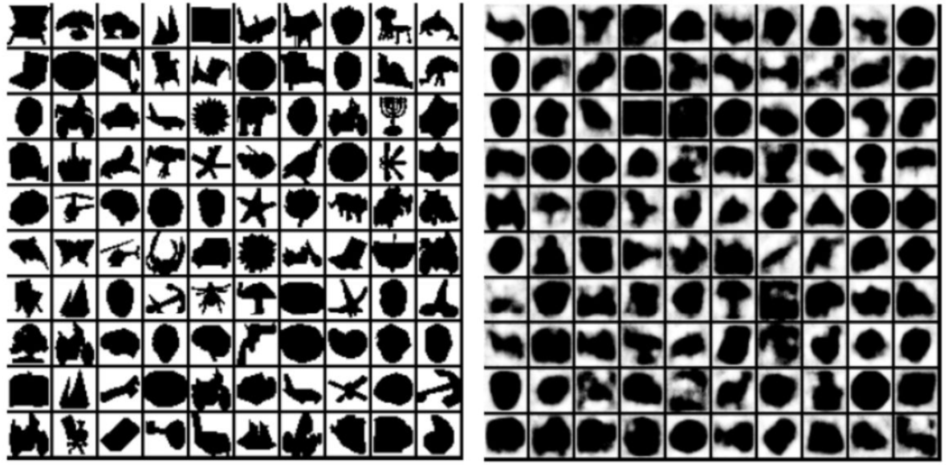
.

Cannot be done in parallel.



FVSBN

Example

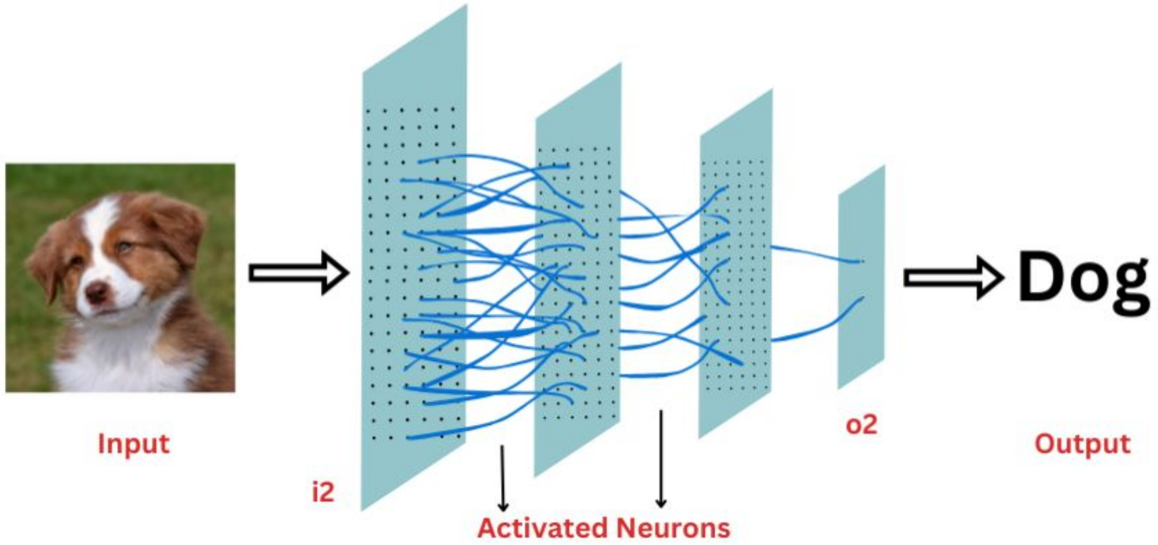


Training data on the left (Caltech 101 Silhouettes). Samples from the model on the right.

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

Deep learning

Classification

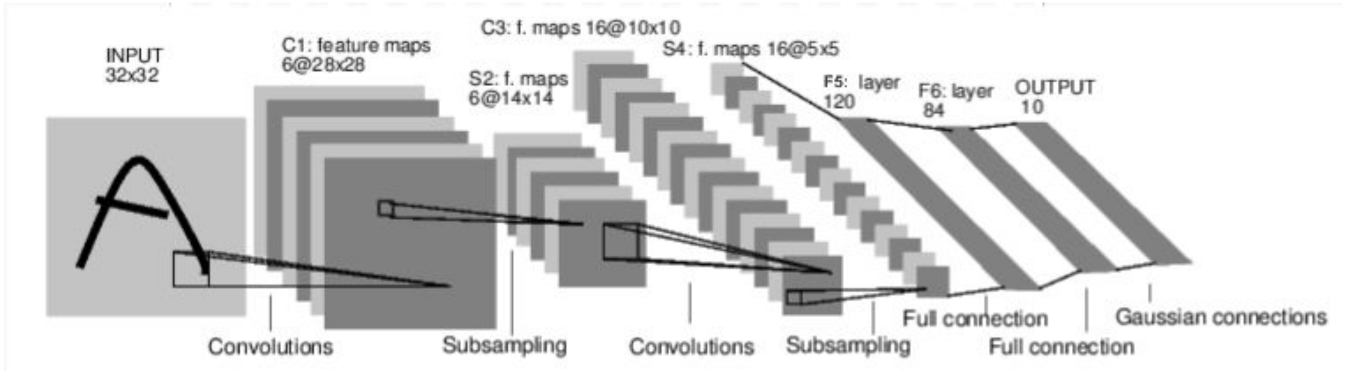


logistic regression

$$f(x) = \sigma(\theta^\top x) = \frac{1}{1 + \exp(-\theta^\top x)}$$

- Train with gradient descent
- Is only linear. Can we make this non-linear?
- **Deep learning**: Apply gradient descent on a non-linear function of **x**, implemented by neural networks.

Deep Neural Networks



Convolutional Network (ConvNet / CNN)

- Many layers with non-linear activations (ReLU)
- Last layer: soft-max (generalization of logistic regression to k classes):

$$P(y^{(i)} = k | x^{(i)}; \theta) = \frac{\exp(\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x^{(i)})}$$

Loss

Maximum log-likelihood:

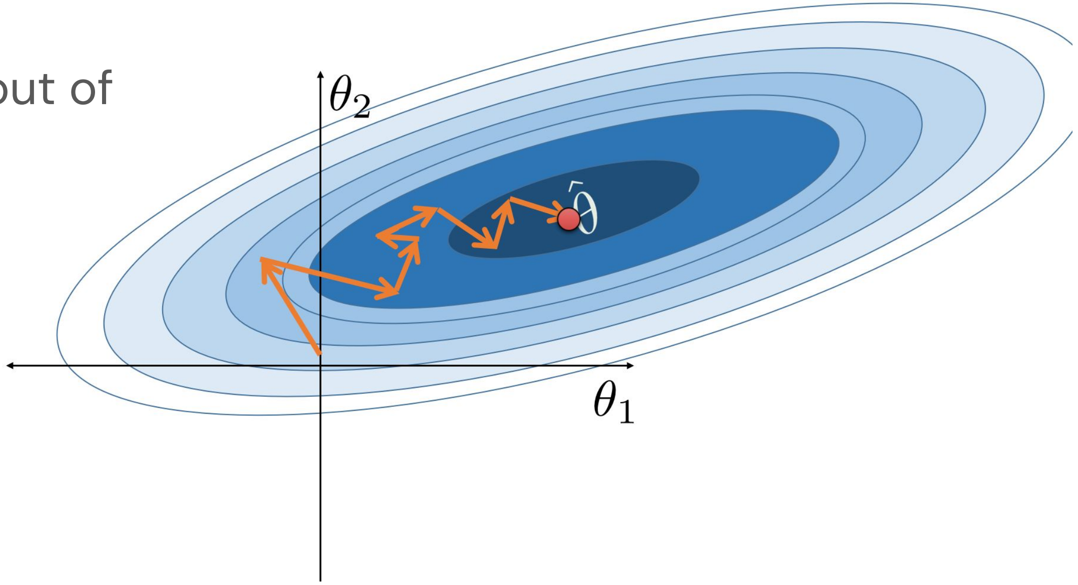
$$J(\theta) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\left\{y^{(i)} = k\right\} \log \frac{\exp(\theta^{(k)\top} x^{(i)})}{\sum_{j=1}^{K} \exp(\theta^{(j)\top} x^{(i)})}\right]$$

(for linear case)

For deep NNs: x is the output before the last layer

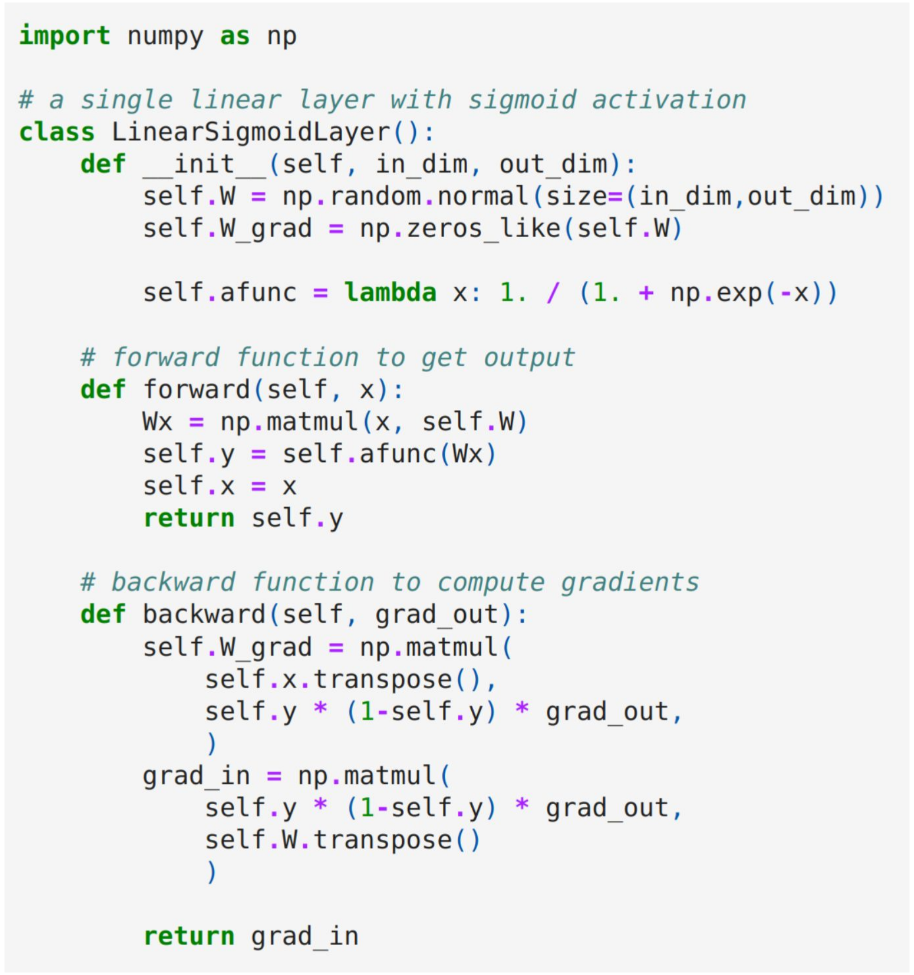
Stochastic gradient descent

- Don’t go over all training data for just one gradient step
- Select a random mini-batch
- Can also help in getting out of local minima



Autodiff

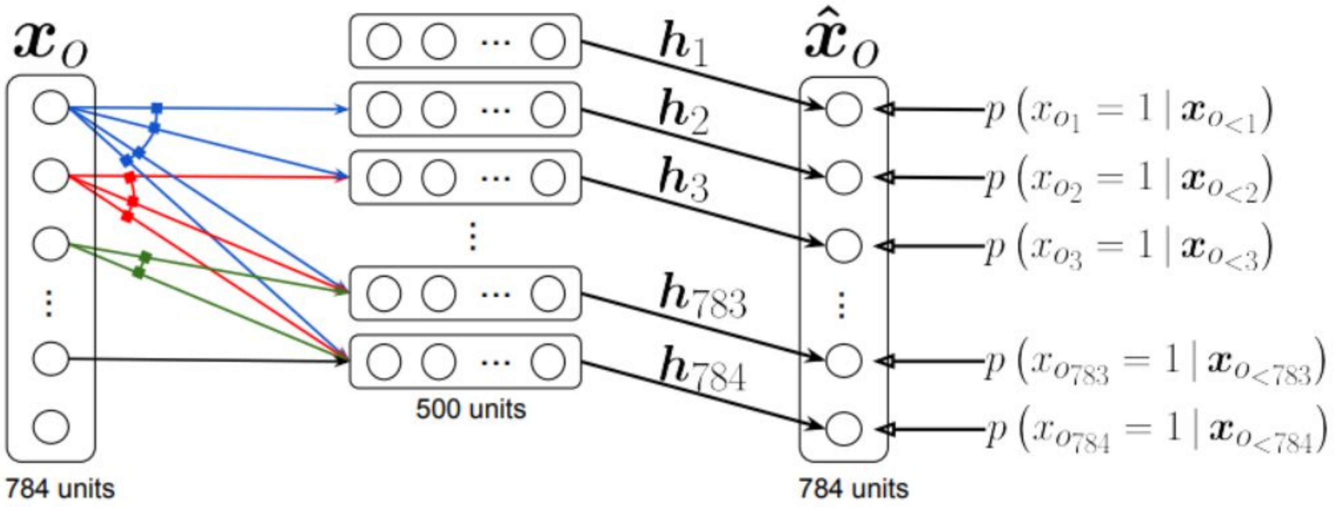
- Many libraries to compute backpropagation automatically
  - Tensorflow, PyTorch, JAX
- Modular components to easily implement and train very complex neural networks.



```python
import numpy as np

# a single linear layer with sigmoid activation
class LinearSigmoidLayer():
    def __init__(self, in_dim, out_dim):
        self.W = np.random.normal(size=(in_dim,out_dim))
        self.W_grad = np.zeros_like(self.W)

        self.afunc = lambda x: 1. / (1. + np.exp(-x))

    # forward function to get output
    def forward(self, x):
        Wx = np.matmul(x, self.W)
        self.y = self.afunc(Wx)
        self.x = x
        return self.y

    # backward function to compute gradients
    def backward(self, grad_out):
        self.W_grad = np.matmul(
            self.x.transpose(),
            self.y * (1-self.y) * grad_out,
            )
        grad_in = np.matmul(
            self.y * (1-self.y) * grad_out,
            self.W.transpose()
            )

        return grad_in
```

NADE: Neural Autoregressive Density Estimation

Similar to FVSBN, but with one hidden layer:



$\boldsymbol{x}_o$
$\hat{\boldsymbol{x}}_o$
$\boldsymbol{h}_1$
$\boldsymbol{h}_2$
$\boldsymbol{h}_3$
$\boldsymbol{h}_{783}$
$\boldsymbol{h}_{784}$
$p(x_{o_1} = 1 \mid \boldsymbol{x}_{o_{<1}})$
$p(x_{o_2} = 1 \mid \boldsymbol{x}_{o_{<2}})$
$p(x_{o_3} = 1 \mid \boldsymbol{x}_{o_{<3}})$
$p(x_{o_{783}} = 1 \mid \boldsymbol{x}_{o_{<783}})$
$p(x_{o_{784}} = 1 \mid \boldsymbol{x}_{o_{<784}})$
784 units
500 units
784 units

2011

NADE results:

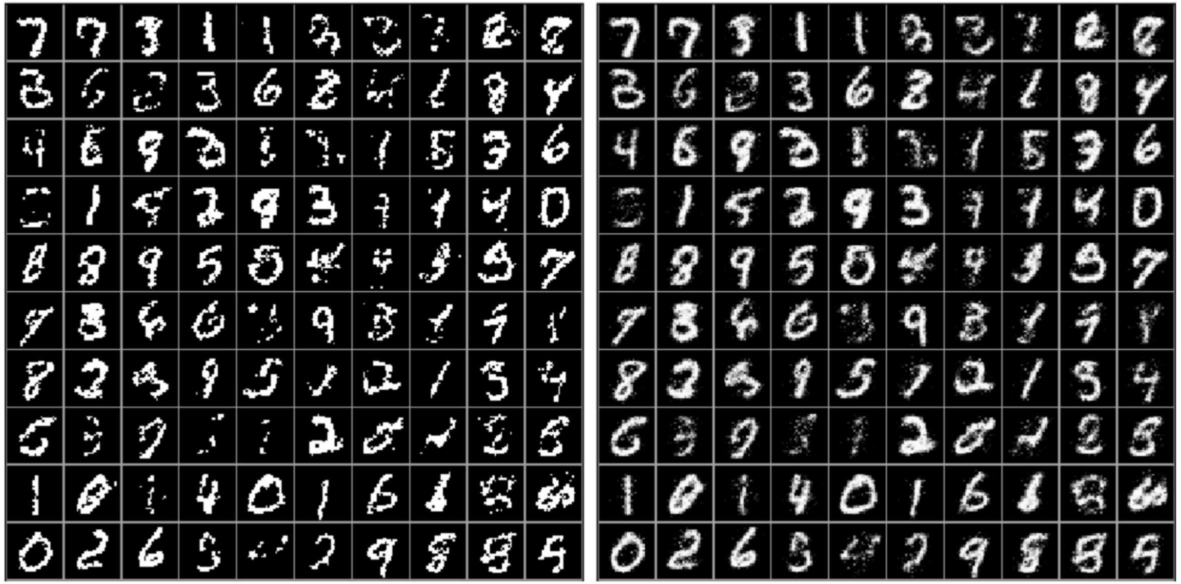
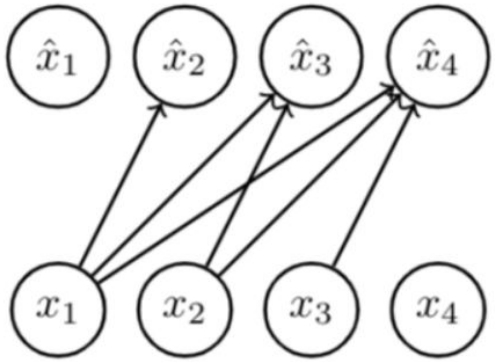
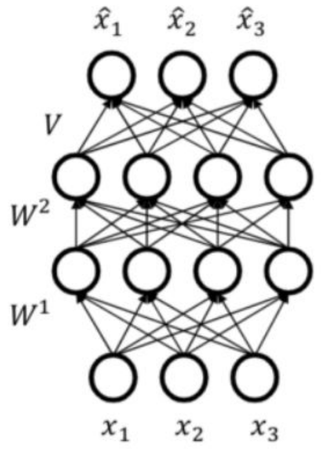


Figure 4: **(Left)**: samples from NADE trained on binarized MNIST. **(Right)**: probabilities from which each pixel was sampled. Ancestral sampling was used with the same fixed ordering used during training.

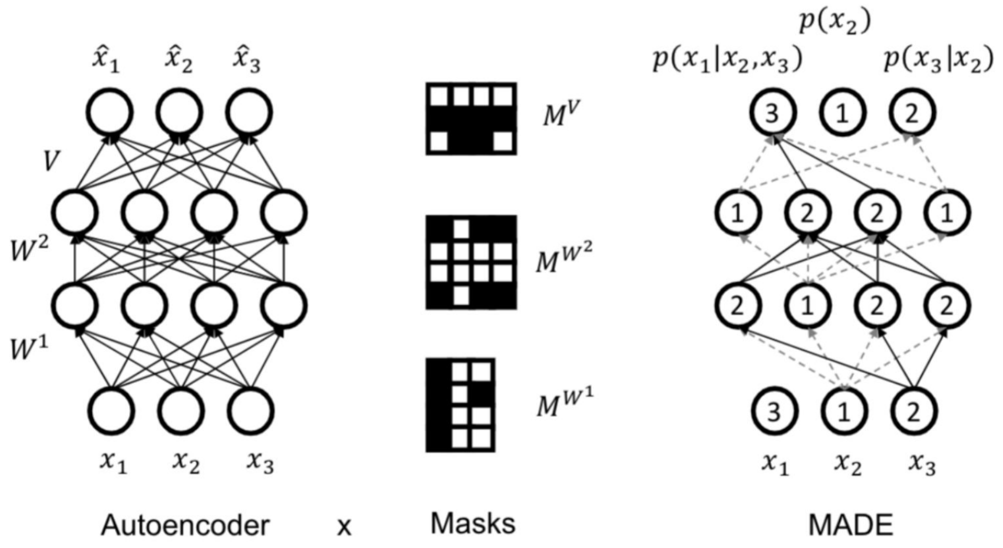
autoregressive vs. autoencoders





- An autoencoder is not a generative model
- An autoregressive model is like an autoencoder that predicts a shifted input

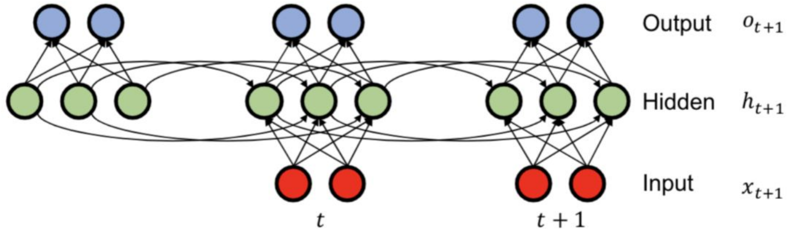
MADE: Masked Autoencoder for Distribution Estimation



Use masks that imply a DAG structure

Recurrent Neural Networks RNNs

Challenge: model $p(x_t | x_{1:t-1}; \boldsymbol{\alpha}^t)$. "History" $x_{1:t-1}$ keeps getting longer.
Idea: keep a summary and recursively update it



$$\begin{aligned}\text{Summary update rule:} \quad h_{t+1} &= tanh(W_{hh} h_t + W_{xh} x_{t+1}) \\ \text{Prediction:} \quad o_{t+1} &= W_{hy} h_{t+1} \\ \text{Summary initalization:} \quad h_0 &= \boldsymbol{b}_0\end{aligned}$$

RNNs as autoregressive models

Pros:

- Can be applied to sequences of arbitrary length.
- Very general: For every computable function, there exists a finite RNN that can compute it

Cons:

- Still requires an ordering
- Sequential likelihood evaluation (very slow for training)
- Sequential generation (unavoidable in an autoregressive model)
- Can be difficult to train (vanishing/exploding gradients)

Example: pixelRNN

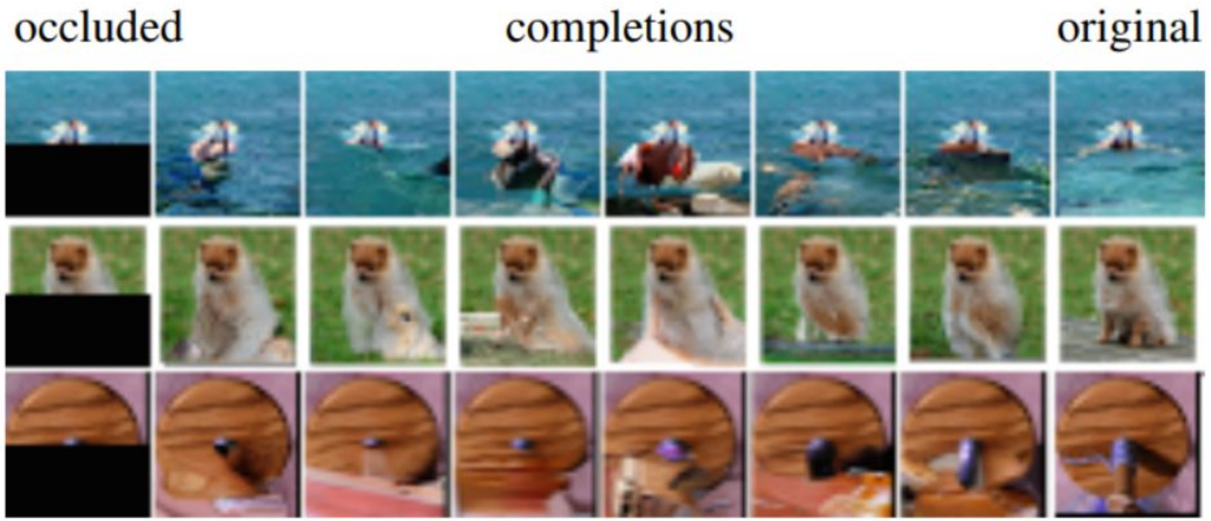


Figure 1. Image completions sampled from a PixelRNN.

PixelCNN

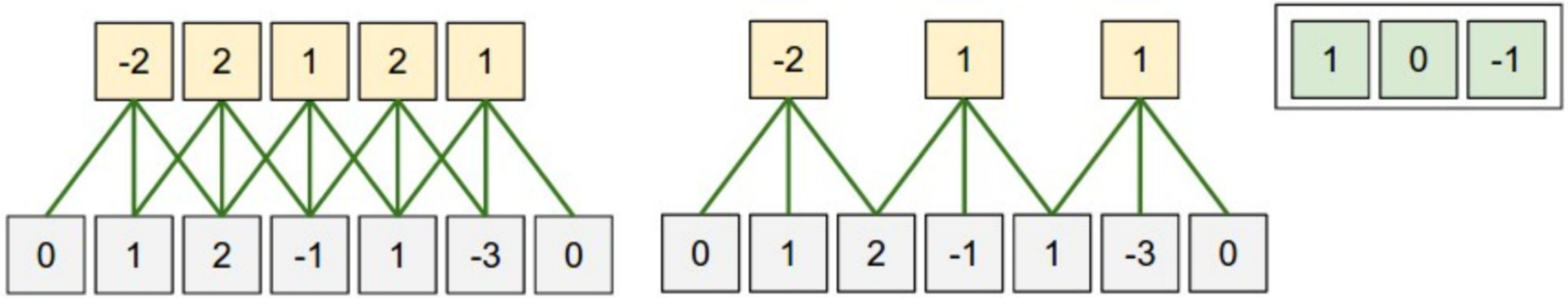
An autoregressive model of images using convolutions.

Convolutions

- Have proven to be a very powerful inductive bias in vision
- Can we use it for generative models?
- 2 key properties:
  1. Hierarchy based on locality
  2. Shared weights

ConvNets

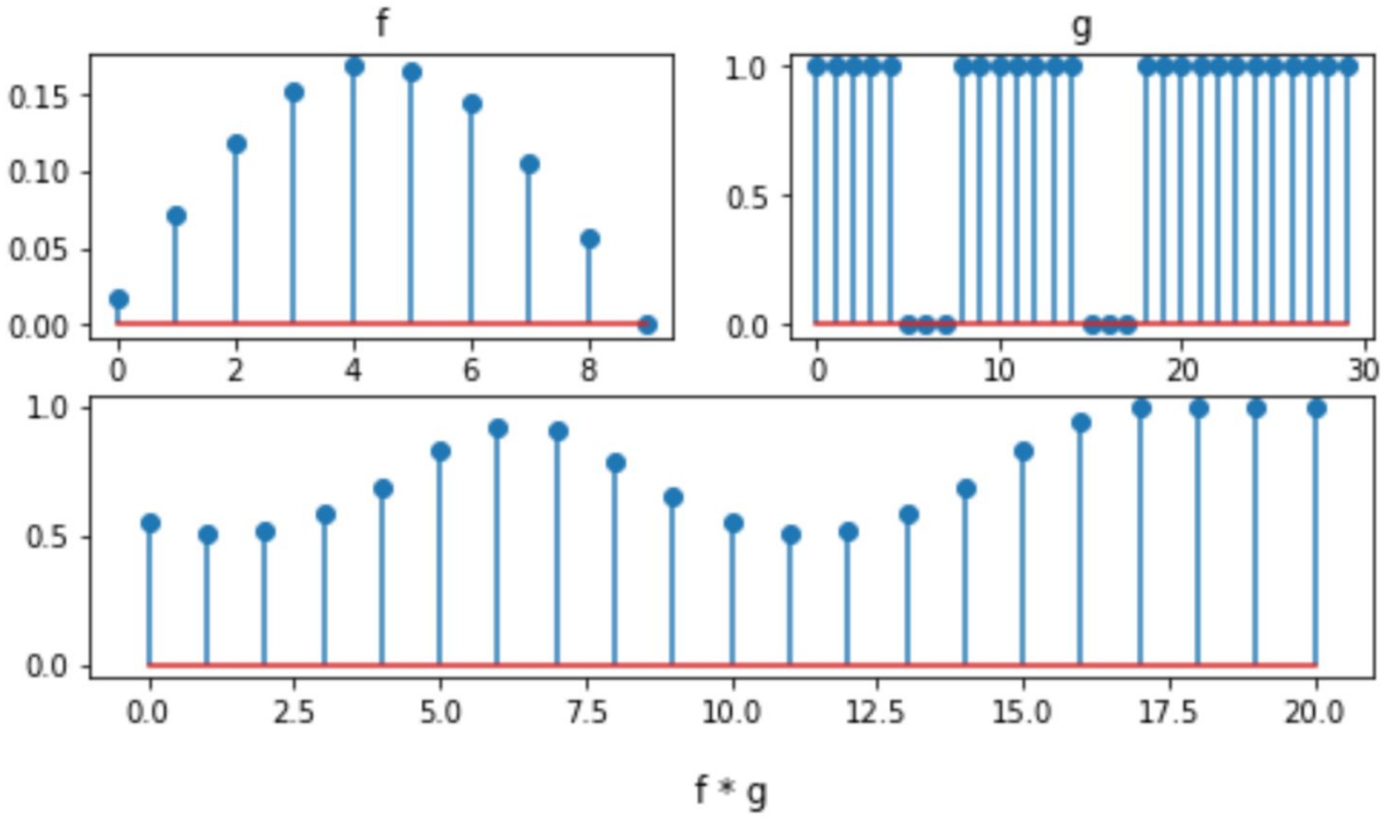
Convolution: $$(f * g)[p] \triangleq \underbrace{\sum_{t=1}^{n} f[t]g[p+t]}_{\text{dot product of } f \text{ with part of } g}$$



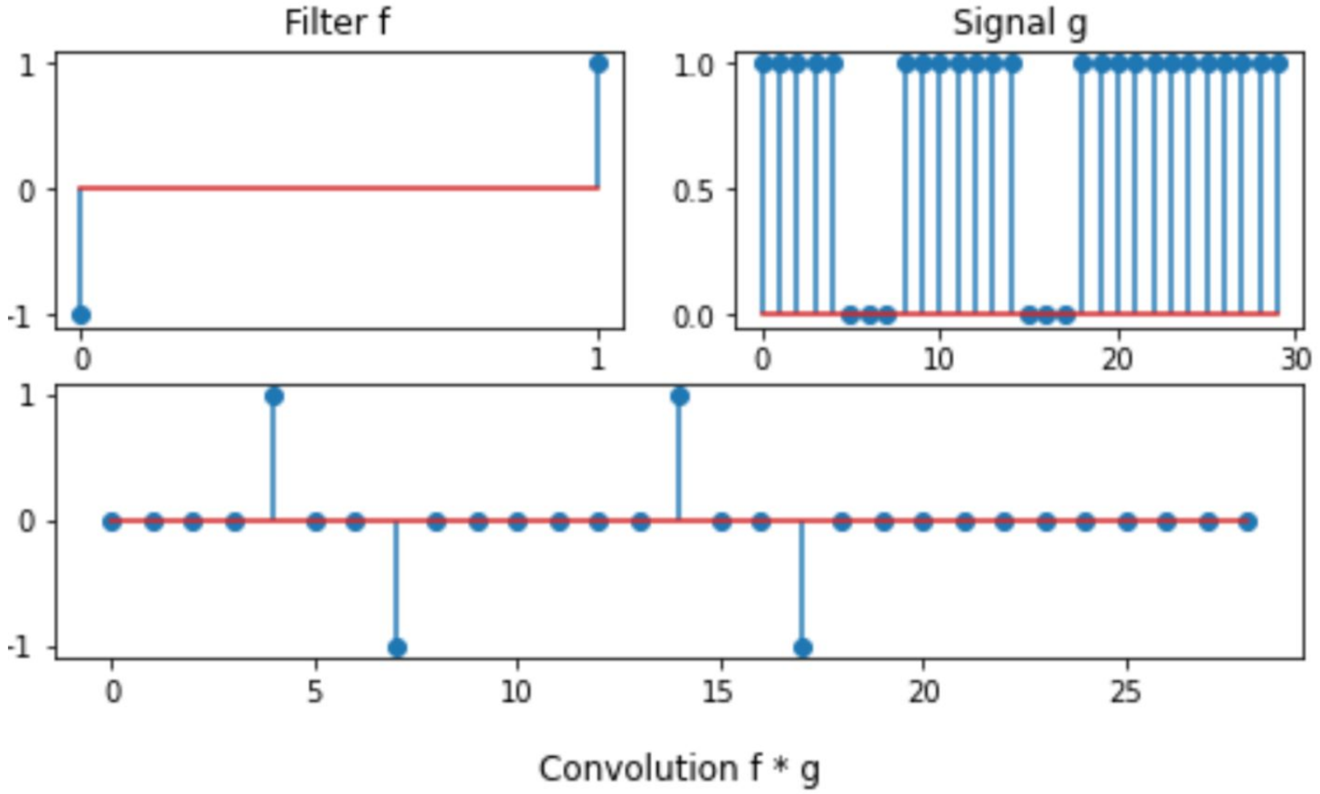
The green sequence `[1,0,-1]` is the filter. The signal g is in gray, and the outputs of the convolution are in yellow.

$$(f * g)[p] = 1 \cdot g_p + 0 \cdot g_{p+1} - 1 \cdot g_{p+2}$$

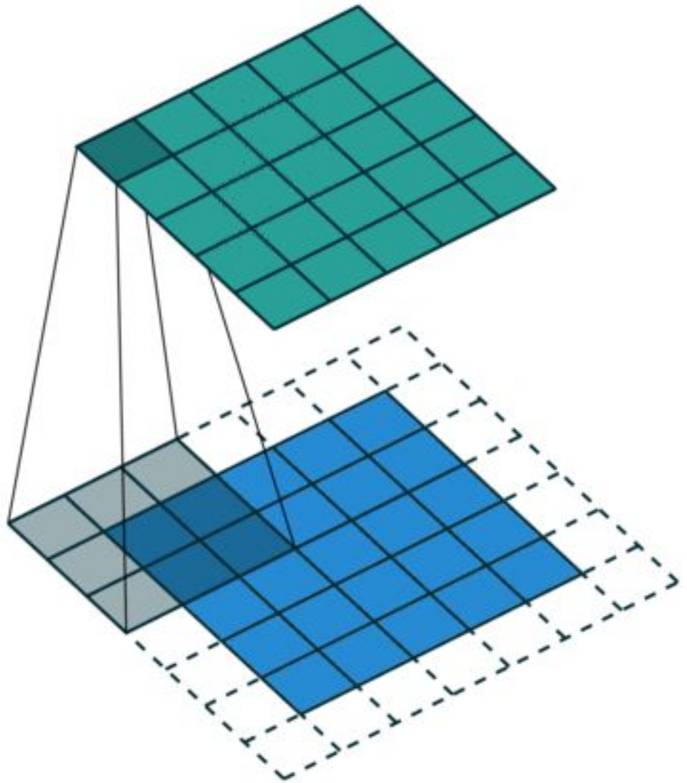
Smoothing


f
g
0.15
0.10
0.05
0.00
0
2
4
6
8
1.0
0.5
0.0
0
10
20
30
1.0
0.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
f * g

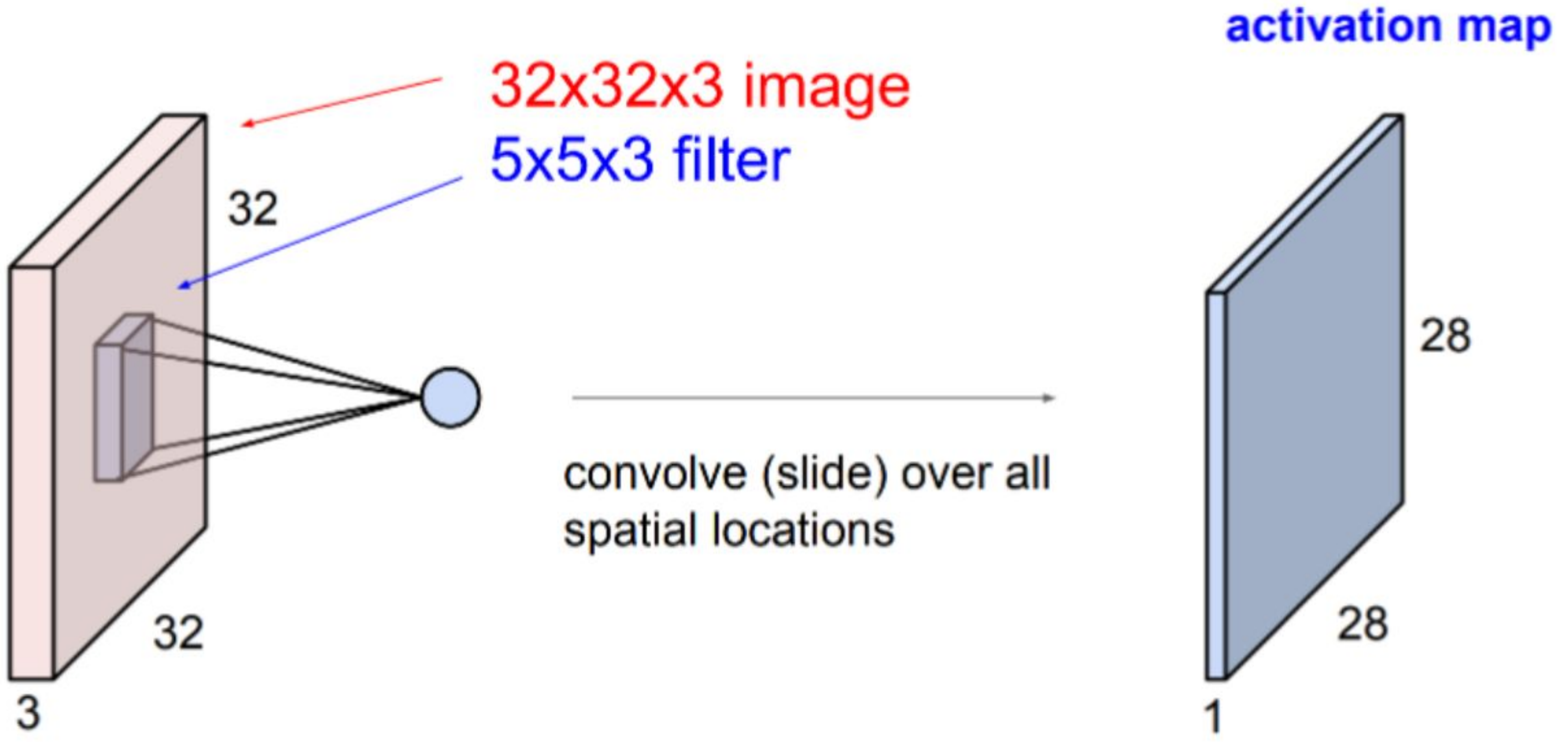
Edge detection



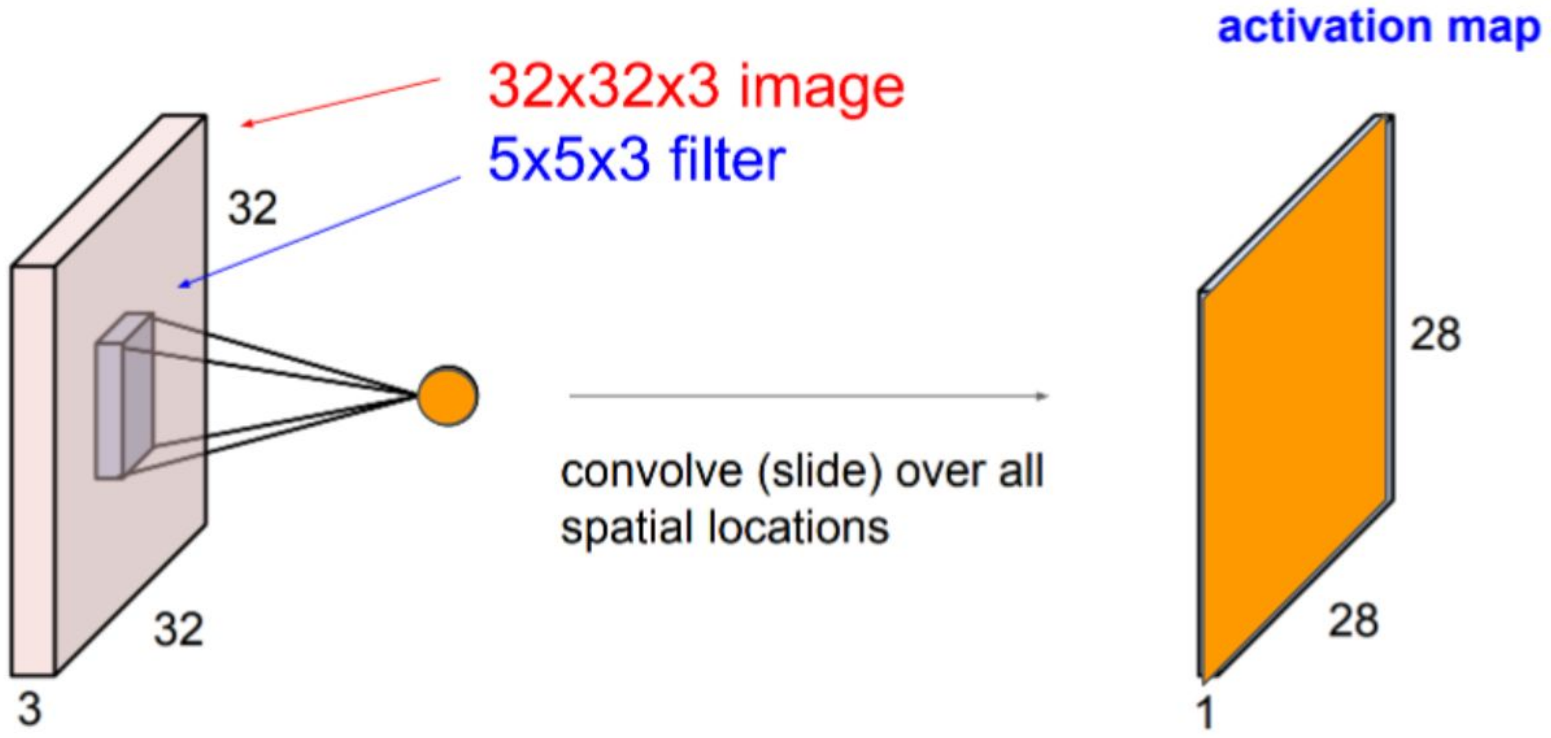
2D convolutions



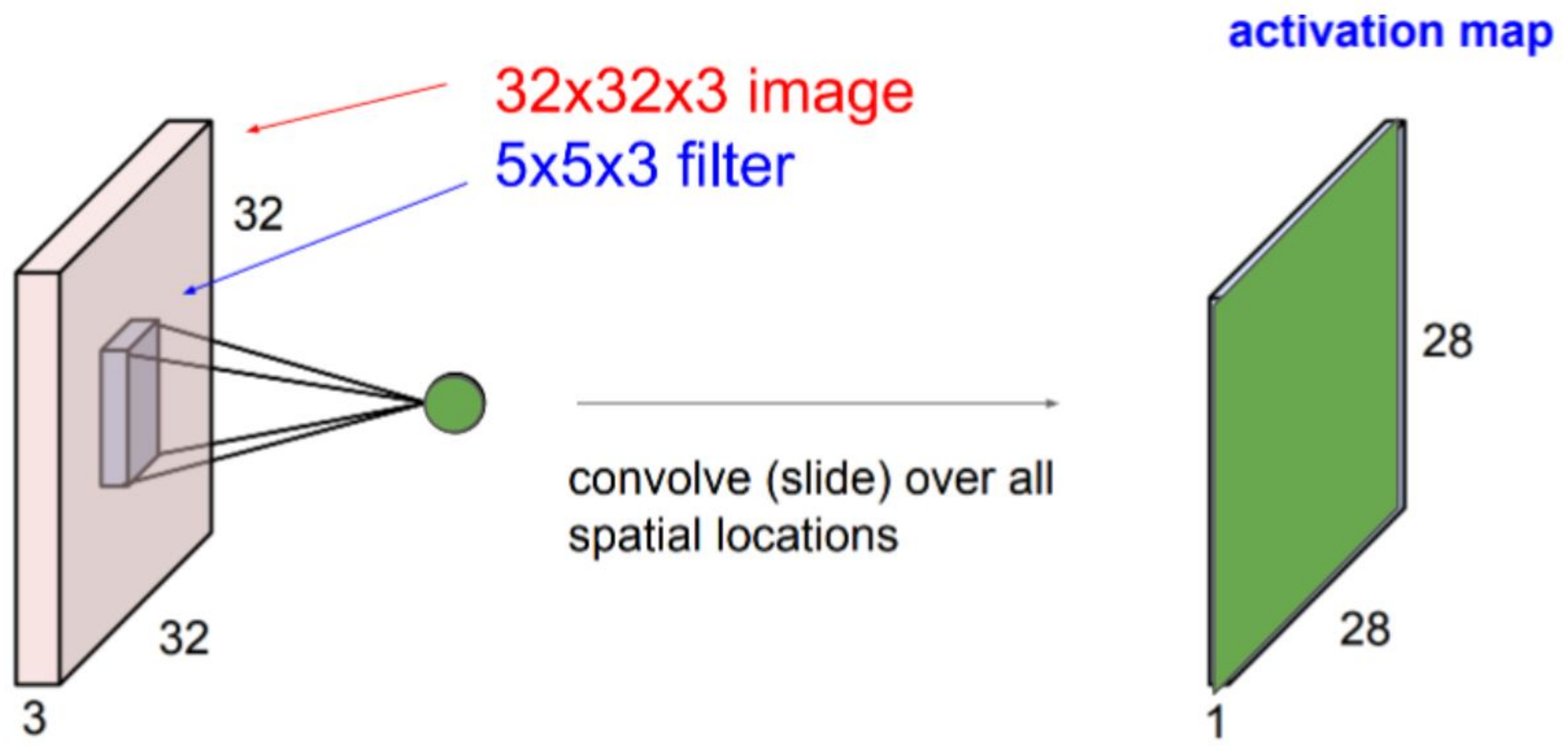
Convolutional layers



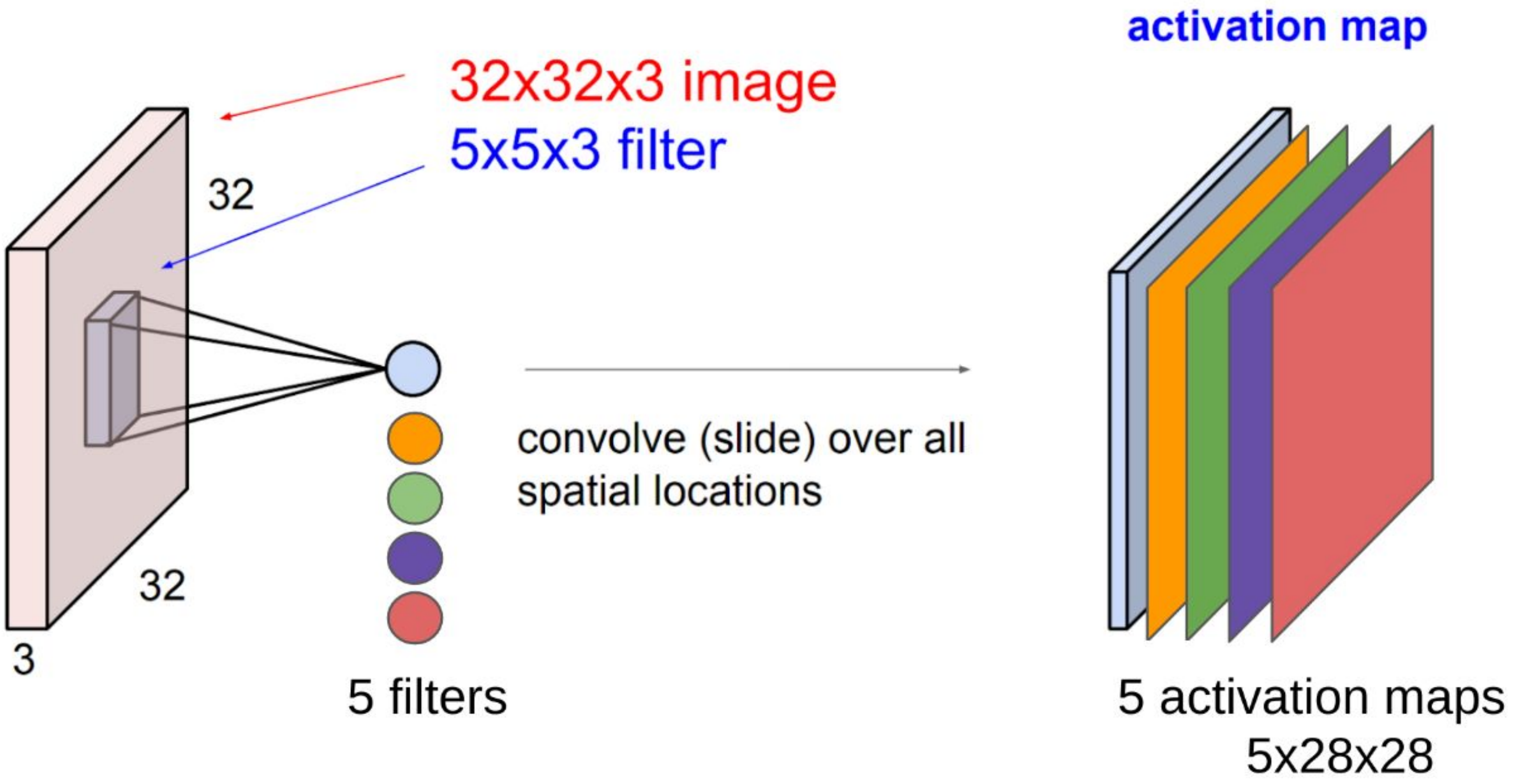
Convolutional layers



Convolutional layers



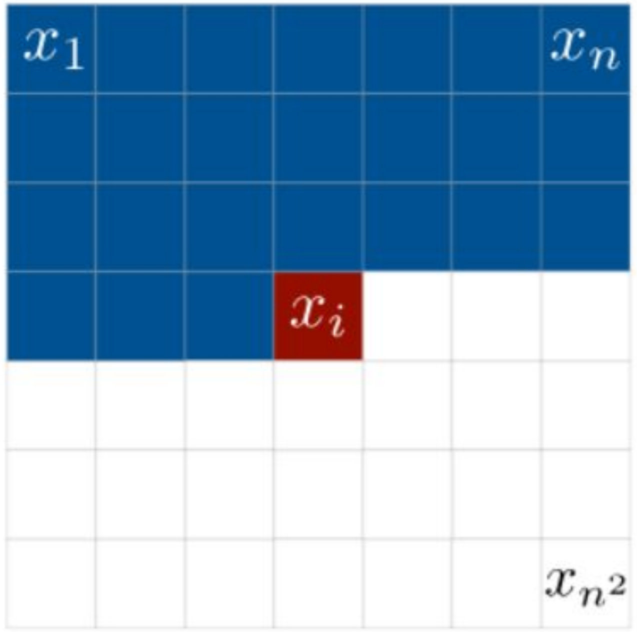
Convolutional layers



PixelCNN

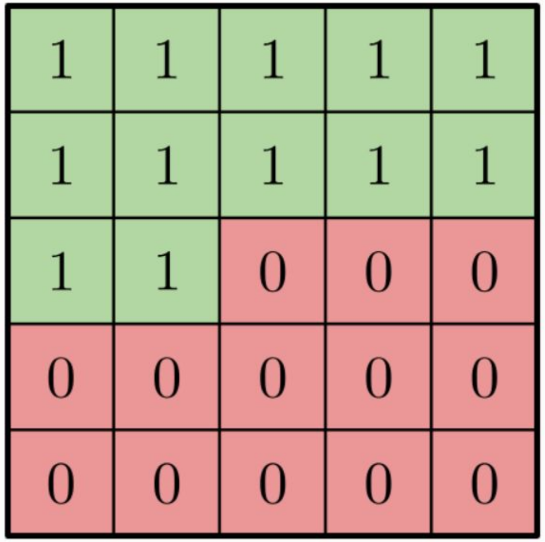
- Efficient implementation of modeling pixel ***x**_i* as a function of all previous pixels.

- Based on convolutions.



Masked convolutions

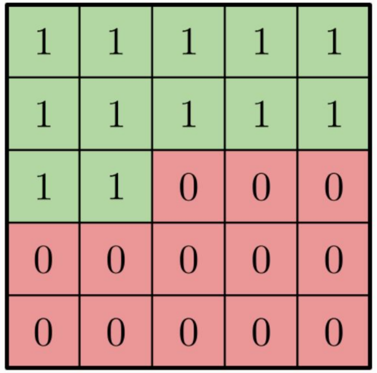
Convolution weights are multiplied by a mask:



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

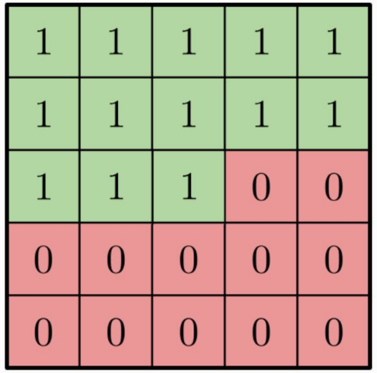
Convolution stack - option 1

First layer:



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

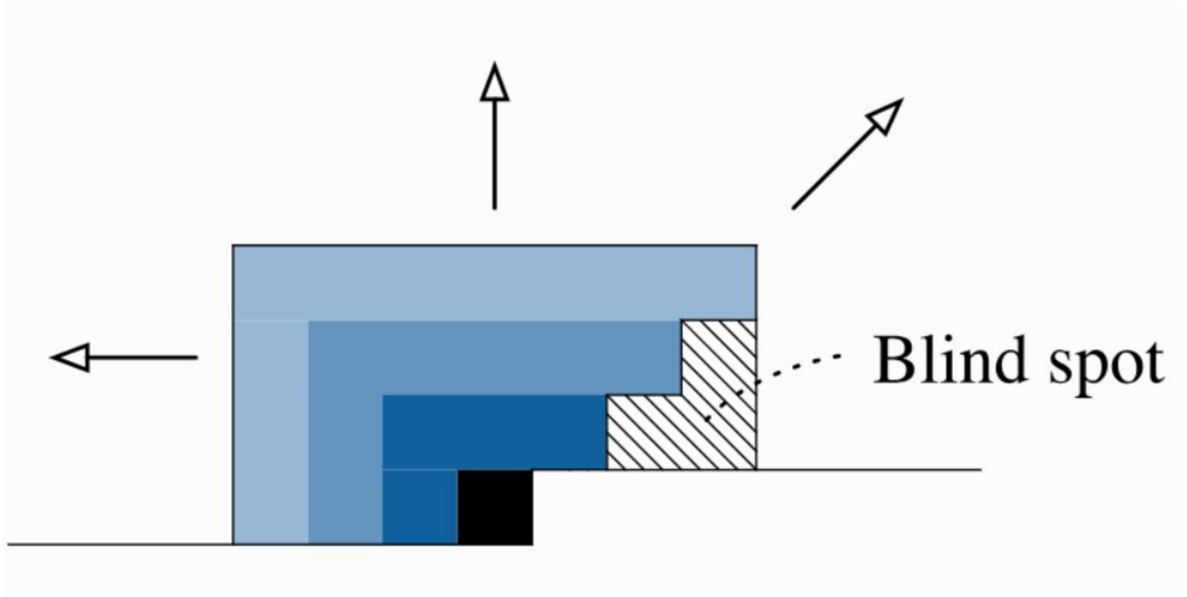
Other layers:



1	1	1	1	1
1	1	1	1	1
1	1	1	0	0
0	0	0	0	0
0	0	0	0	0

problem

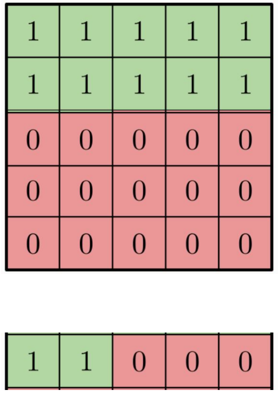
Leads to a “blind spot” in the upper left diagonal



Convolution stack - option 2

Separate vertical and horizontal convolutions.

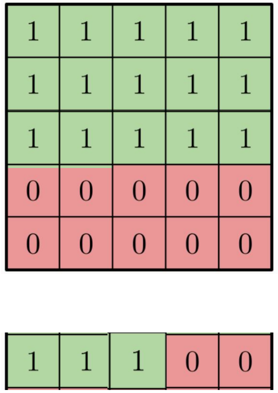
first layer



1	1	1	1	1
1	1	1	1	1
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

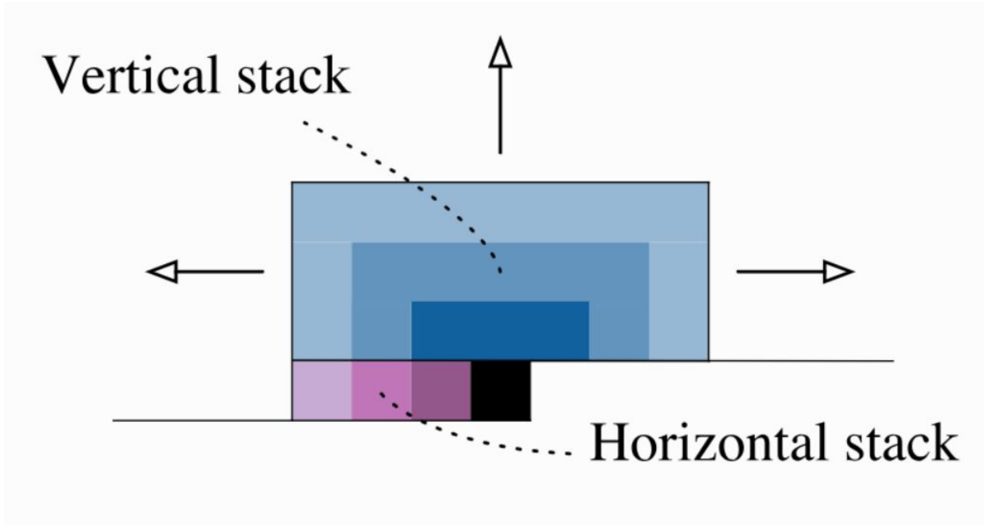
1	1	0	0	0

other layers

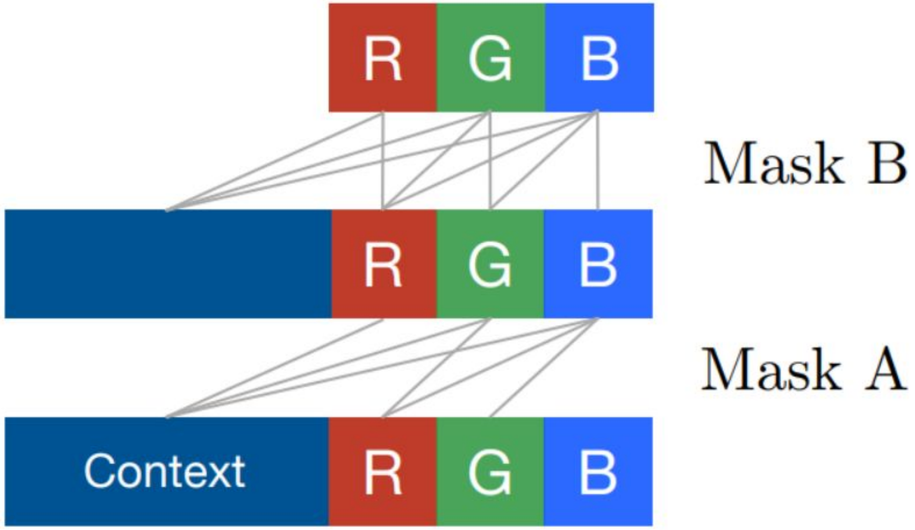


1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
0	0	0	0	0
0	0	0	0	0

1	1	1	0	0



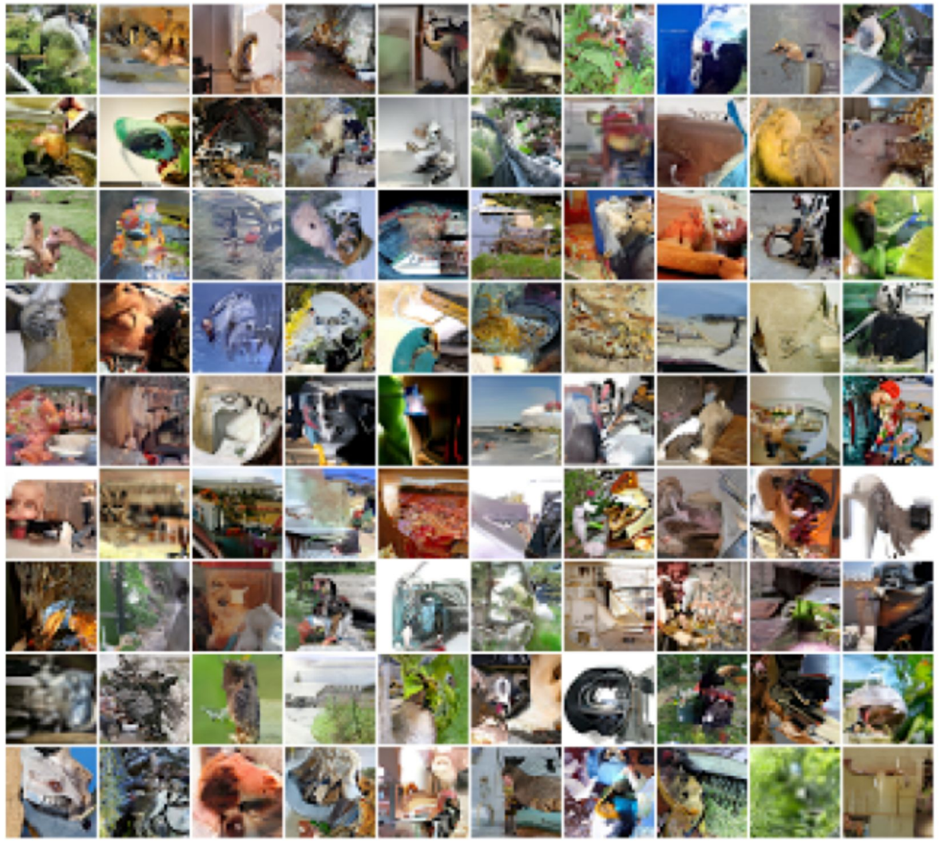
Modeling color



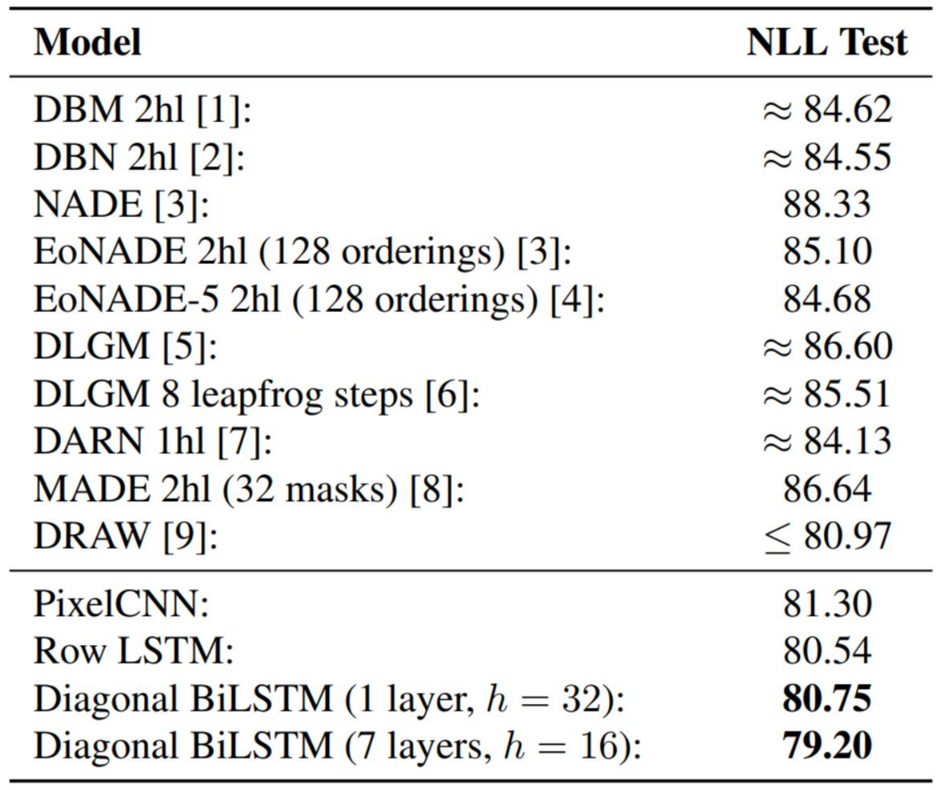
PixelCNN results

Samples from model trained on ImageNet32

(32 x 32 pixels)



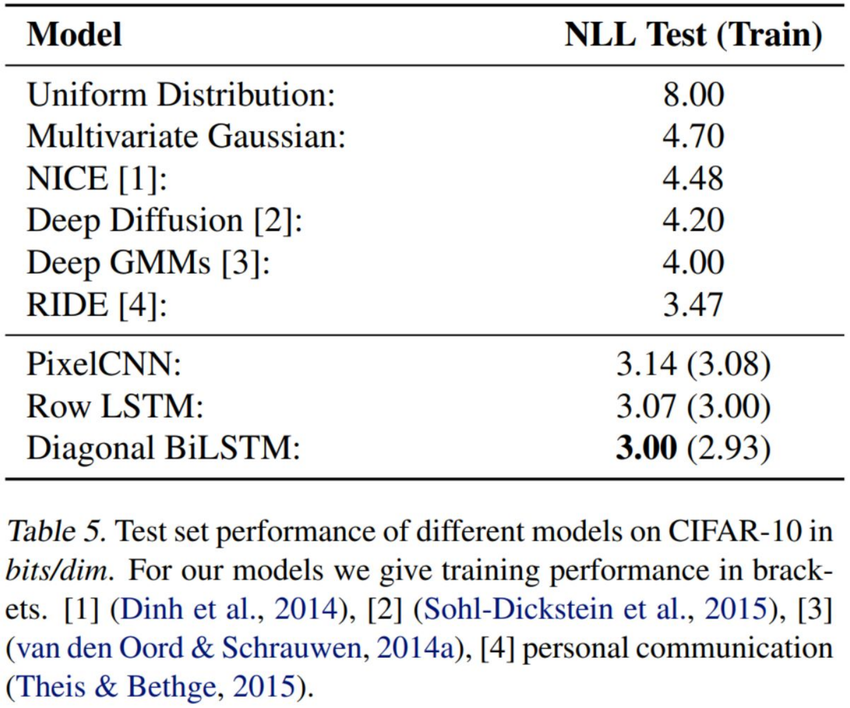
PixelCNN results



Model	NLL Test
DBM 2hl [1]:	≈ 84.62
DBN 2hl [2]:	≈ 84.55
NADE [3]:	88.33
EoNADE 2hl (128 orderings) [3]:	85.10
EoNADE-5 2hl (128 orderings) [4]:	84.68
DLGM [5]:	≈ 86.60
DLGM 8 leapfrog steps [6]:	≈ 85.51
DARN 1hl [7]:	≈ 84.13
MADE 2hl (32 masks) [8]:	86.64
DRAW [9]:	≤ 80.97
PixelCNN:	81.30
Row LSTM:	80.54
Diagonal BiLSTM (1 layer, $h = 32$):	**80.75**
Diagonal BiLSTM (7 layers, $h = 16$):	**79.20**

Table 4. Test set performance of different models on MNIST in *nats* (negative log-likelihood). Prior results taken from [1]

PixelCNN results



Model	**NLL Test (Train)**
Uniform Distribution:	8.00
Multivariate Gaussian:	4.70
NICE [1]:	4.48
Deep Diffusion [2]:	4.20
Deep GMMs [3]:	4.00
RIDE [4]:	3.47
PixelCNN:	3.14 (3.08)
Row LSTM:	3.07 (3.00)
Diagonal BiLSTM:	**3.00** (2.93)

Table 5. Test set performance of different models on CIFAR-10 in *bits/dim*. For our models we give training performance in brackets. [1] (Dinh et al., 2014), [2] (Sohl-Dickstein et al., 2015), [3] (van den Oord & Schrauwen, 2014a), [4] personal communication (Theis & Bethge, 2015).

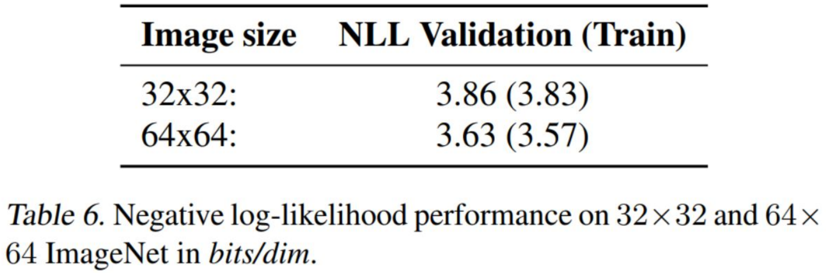
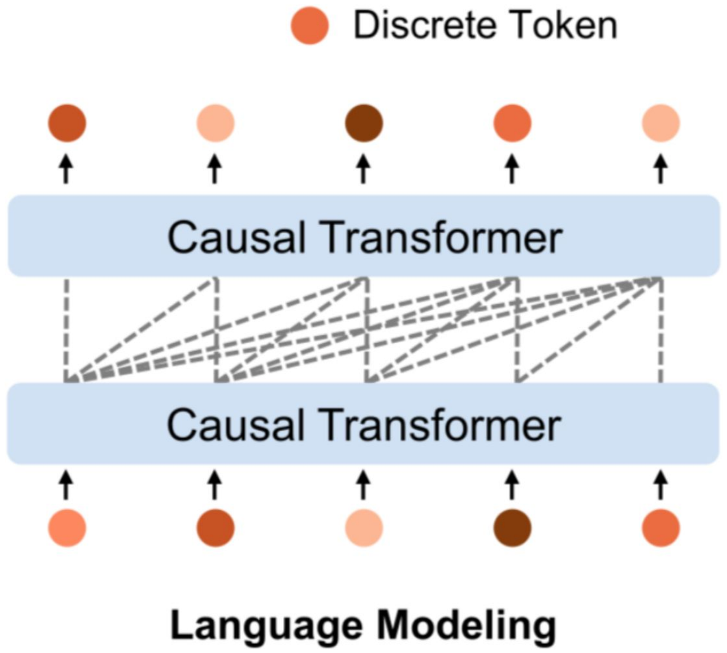


Image size	**NLL Validation (Train)**
32x32:	3.86 (3.83)
64x64:	3.63 (3.57)

Table 6. Negative log-likelihood performance on 32×32 and 64×64 ImageNet in *bits/dim*.

Language Models



Summary of autoregressive models

- Easy to sample from (but can be very long)
- Easy to compute probability: $p(x_1, \cdots, x_{784}) = p(x_1)p(x_2|x_1) \cdots p(x_{784}|x_{1:783})$
- Ideally, can compute all these terms in parallel → important for fast training.
- Can be extended to a continuous representation. E.g:

$$p(x_t \mid x_{<t}) = \mathcal{N}(\mu_\theta(x_{<t}), \Sigma_\theta(x_{<t}))$$

- Not natural for "representation learning". No good global feature.