Introduction to Deep Neural Networks

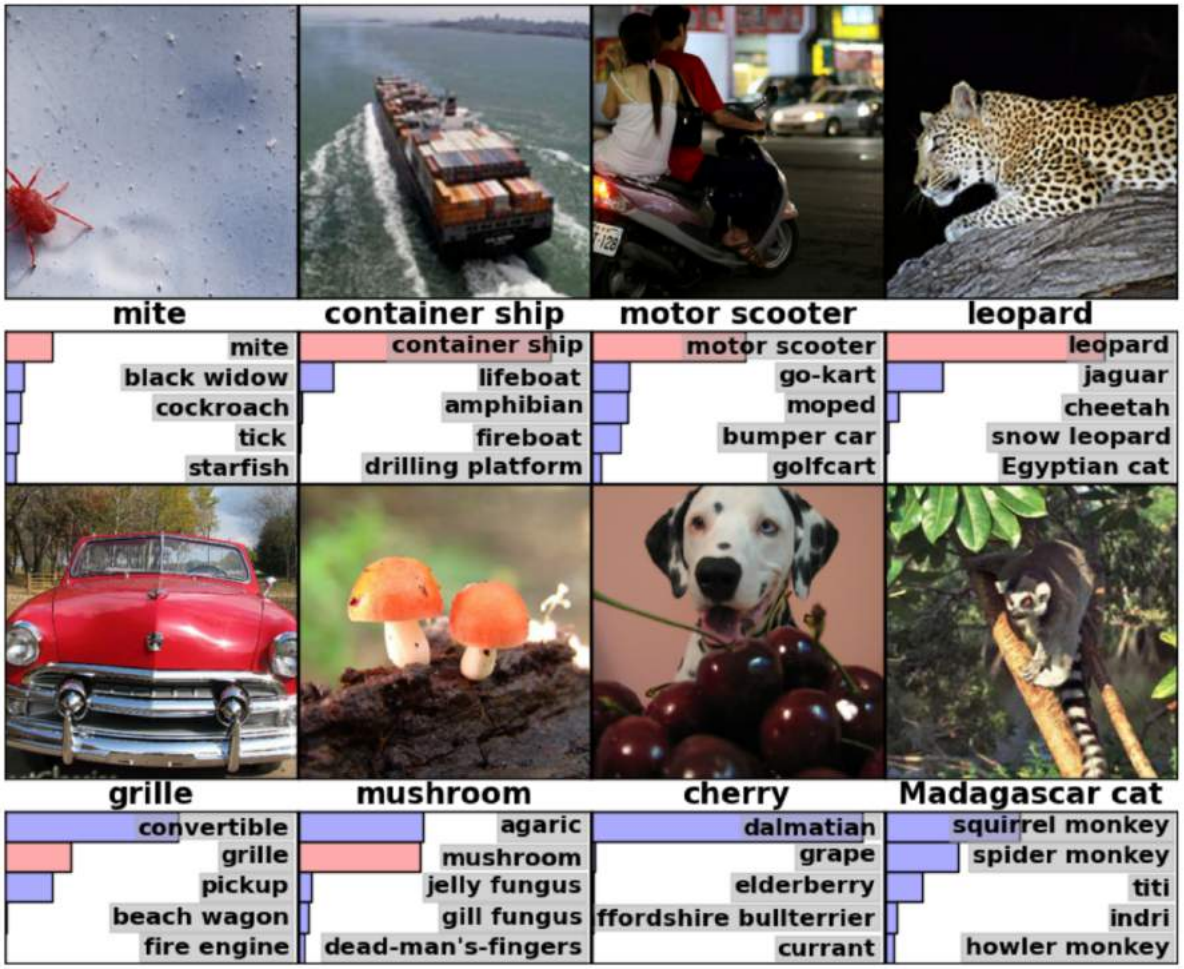
Visual Recognition

Where are the objects of interest?

What are they doing?

What is worth noting about them?

Image Classification



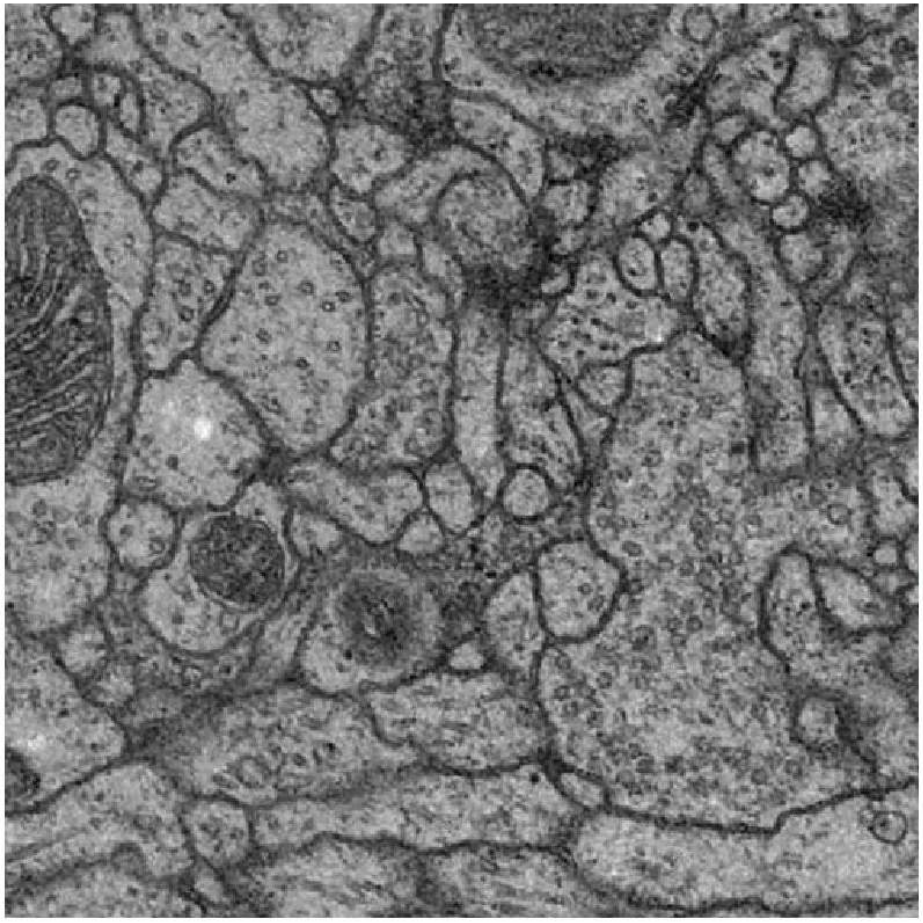
"ImageNet Classification with Deep Convolutional Neural Networks",
Krizhevsky, Sutskever, Hinton*, NIPS,* 2012

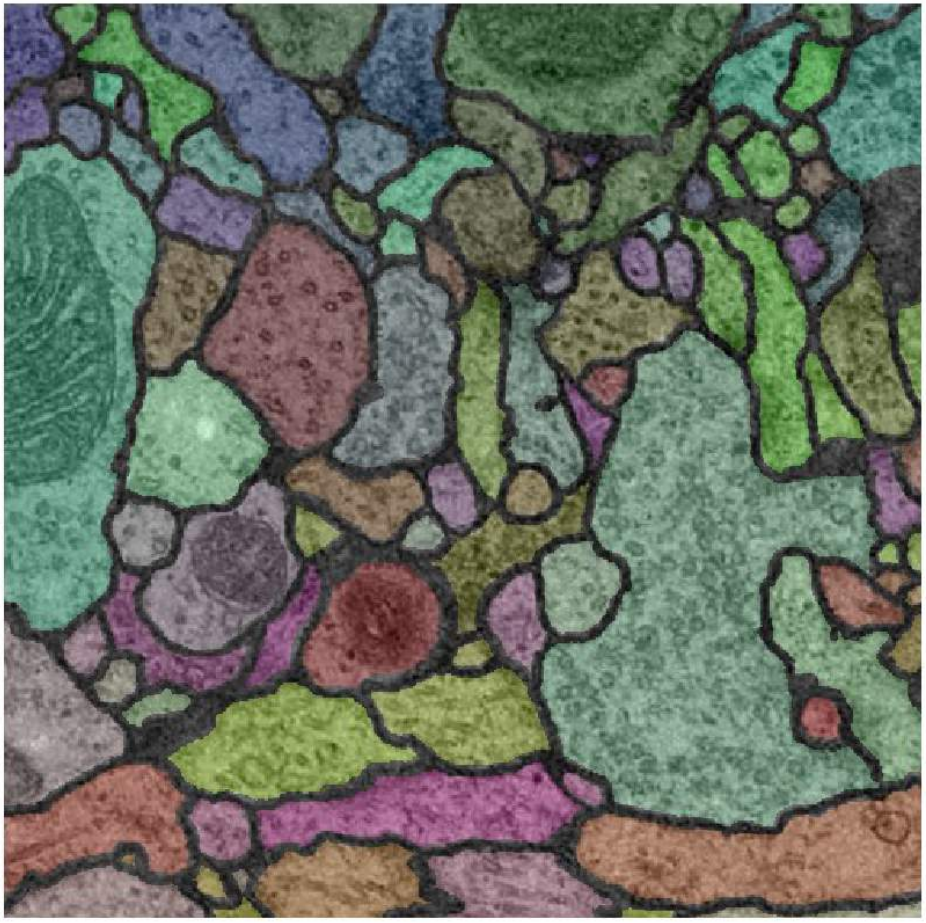
Object Detection and Segmentation

"Mask RCNN", He, Gkioxari, Dollár, Girshick, *ICCV,* 2017

Semantic Segmentation

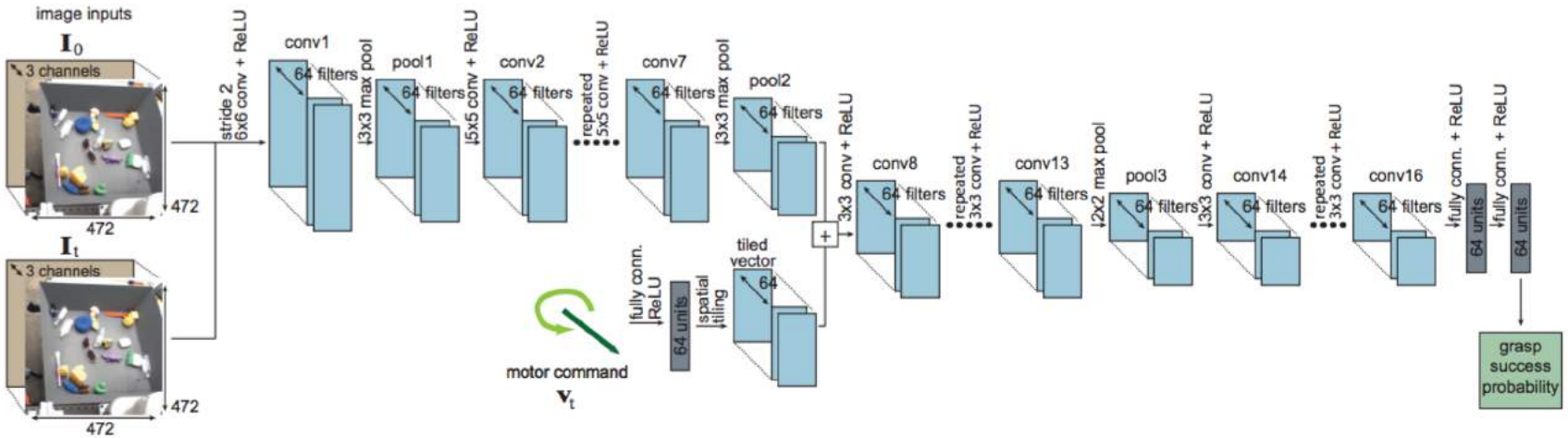
- **Segmentation 3D volumetric images**





Ciresan et al. “DNN segment neuronal membranes...” NIPS 2012
Turaga et al. “Maximin learning of image segmentation” NIPS 2009

Vision based Robot Control



"**Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection**", Levine, Pastor, Krizhevsky, Quillen, *ISER*, 2016

Recognition as Supervised Learning

Image Classification

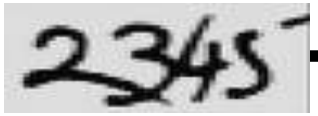
"dog"

classification

Pose Estimation

regression

OCR



"2 3 4 5"

structured prediction

Ranzato

Supervised Learning

$\{x_i, y_i\}$ $\mathrm{i} = 1 \ldots \mathrm{N}$ training dataset

x_i	*i*-th input training example
y_i	*i*-th target label
N	number of training examples

Goal: predict the target label of unseen inputs.

Deep Learning

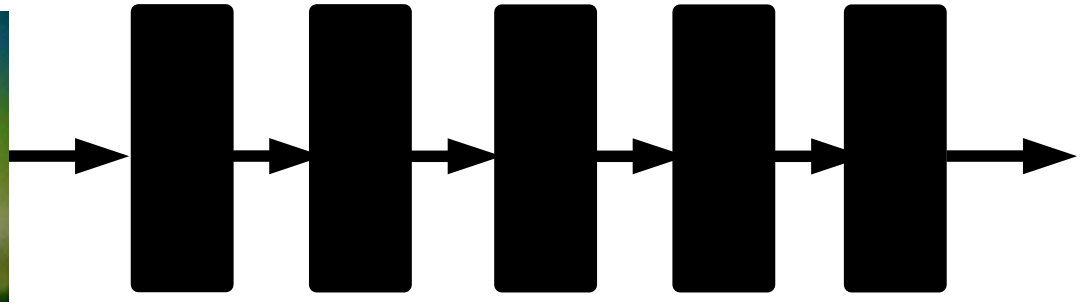
Deep Learning: Composing a set of nonlinear functions g

$$f(x;\theta) = g_1(\, g_2(\, \ldots g_n(x;\theta_n) \ldots ;\theta_2\,);\theta_1\,)$$

- Make predictions by using a sequence of non-linear processing stages
- The resulting intermediate representations can be interpreted as feature hierarchies
- The whole system is jointly learned from data

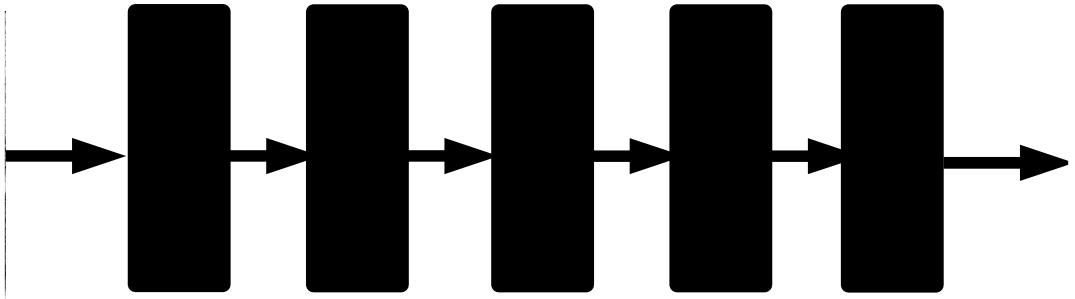
Supervised Deep Learning

Image Classification

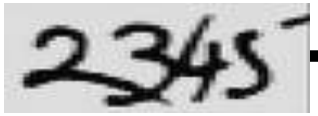


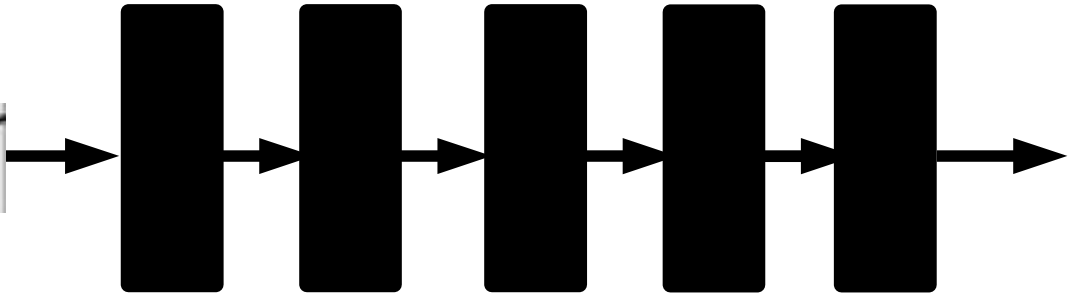
"dog"

Pose Estimation



OCR

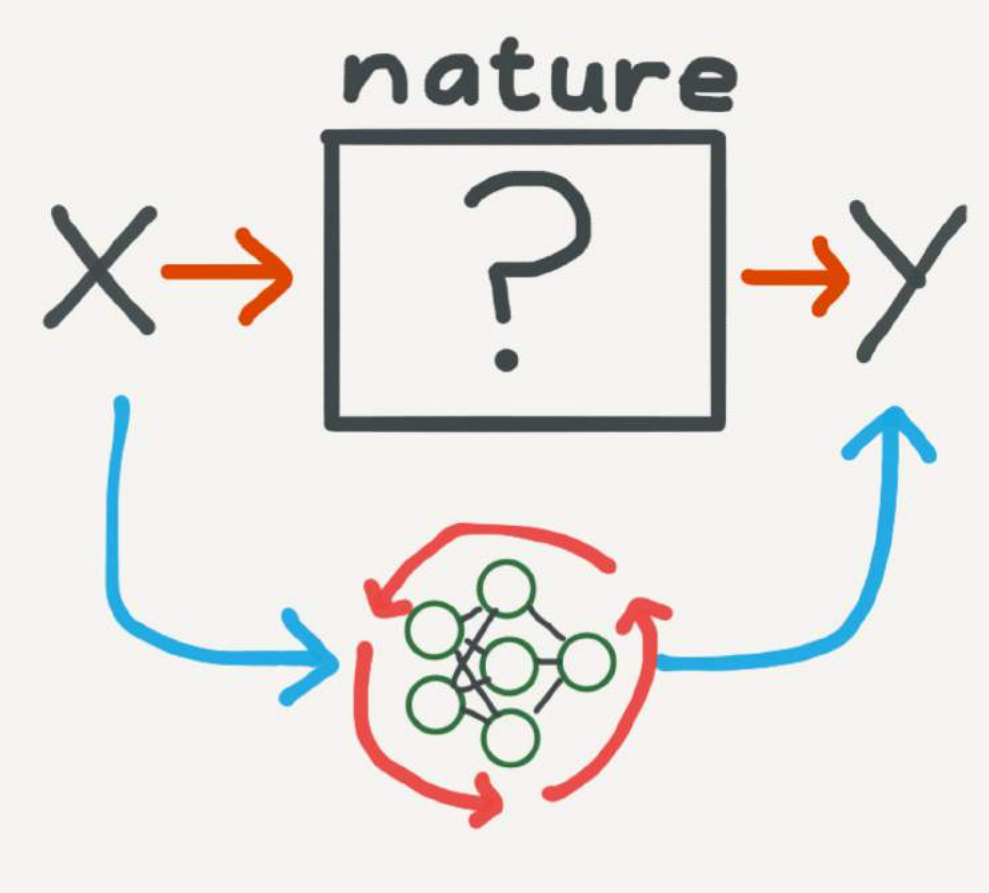




"2 3 4 5"

Ranzato

Supervised Deep Learning



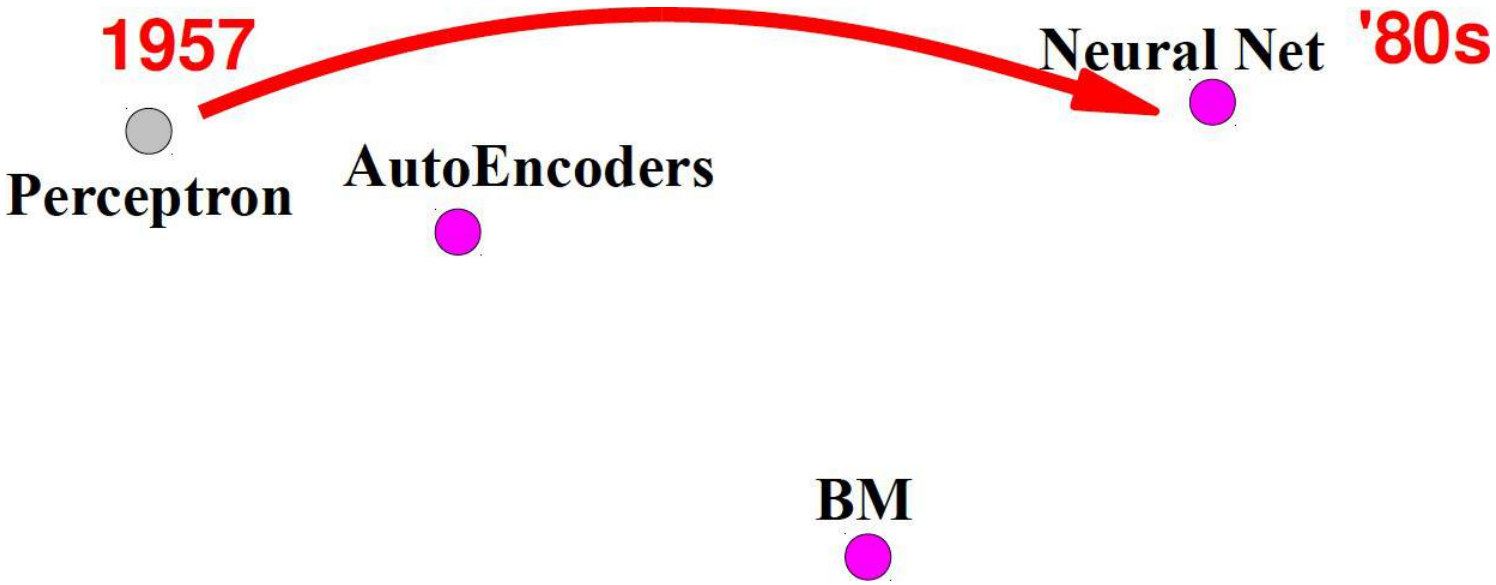
- High capacity models (deep models)
- Large scale datasets (big data)
- Massive computing power (GPUs)

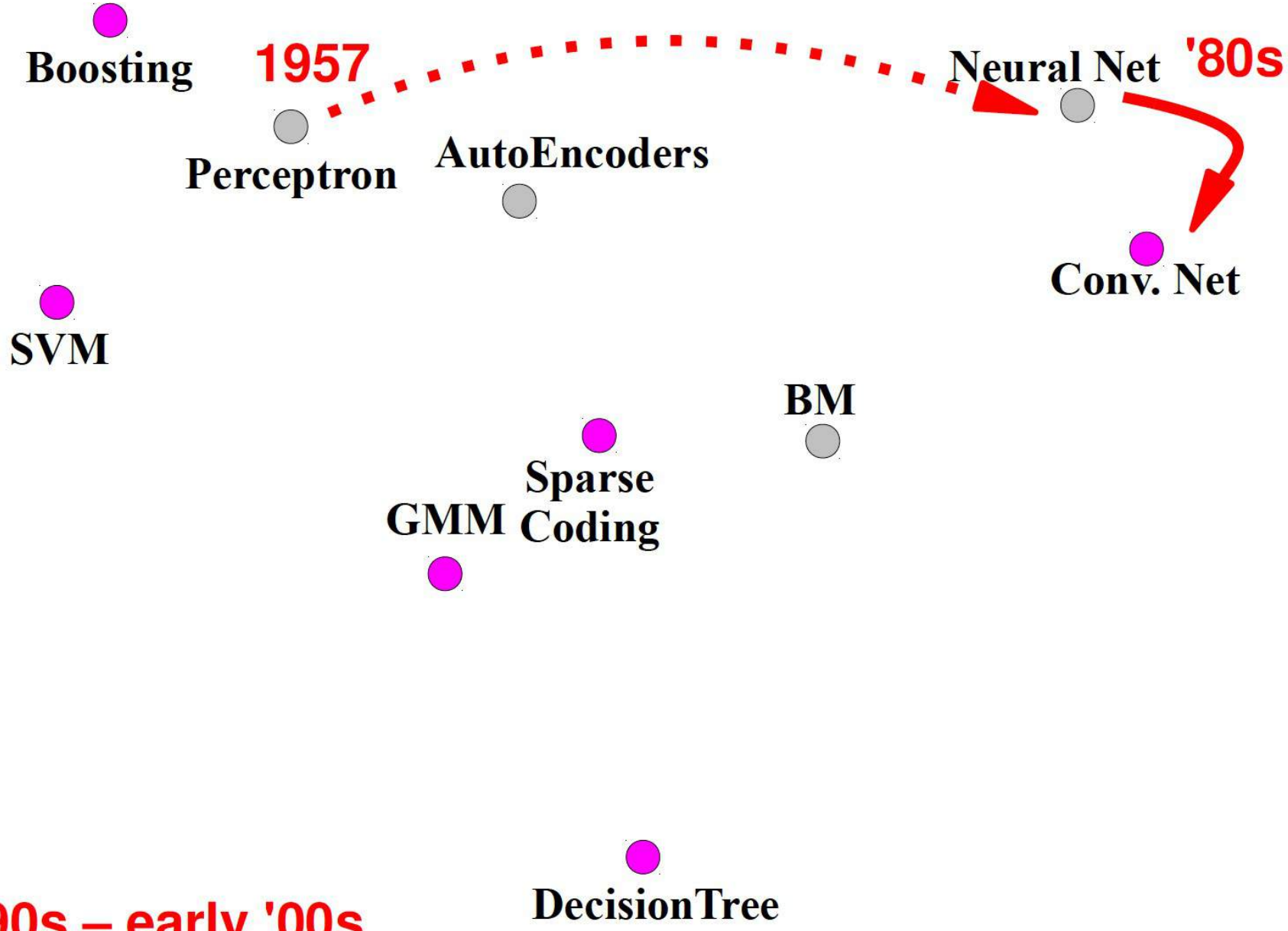
"**Statistical Modeling: The Two Cultures**", Brieman, Statist. Sci., 2001

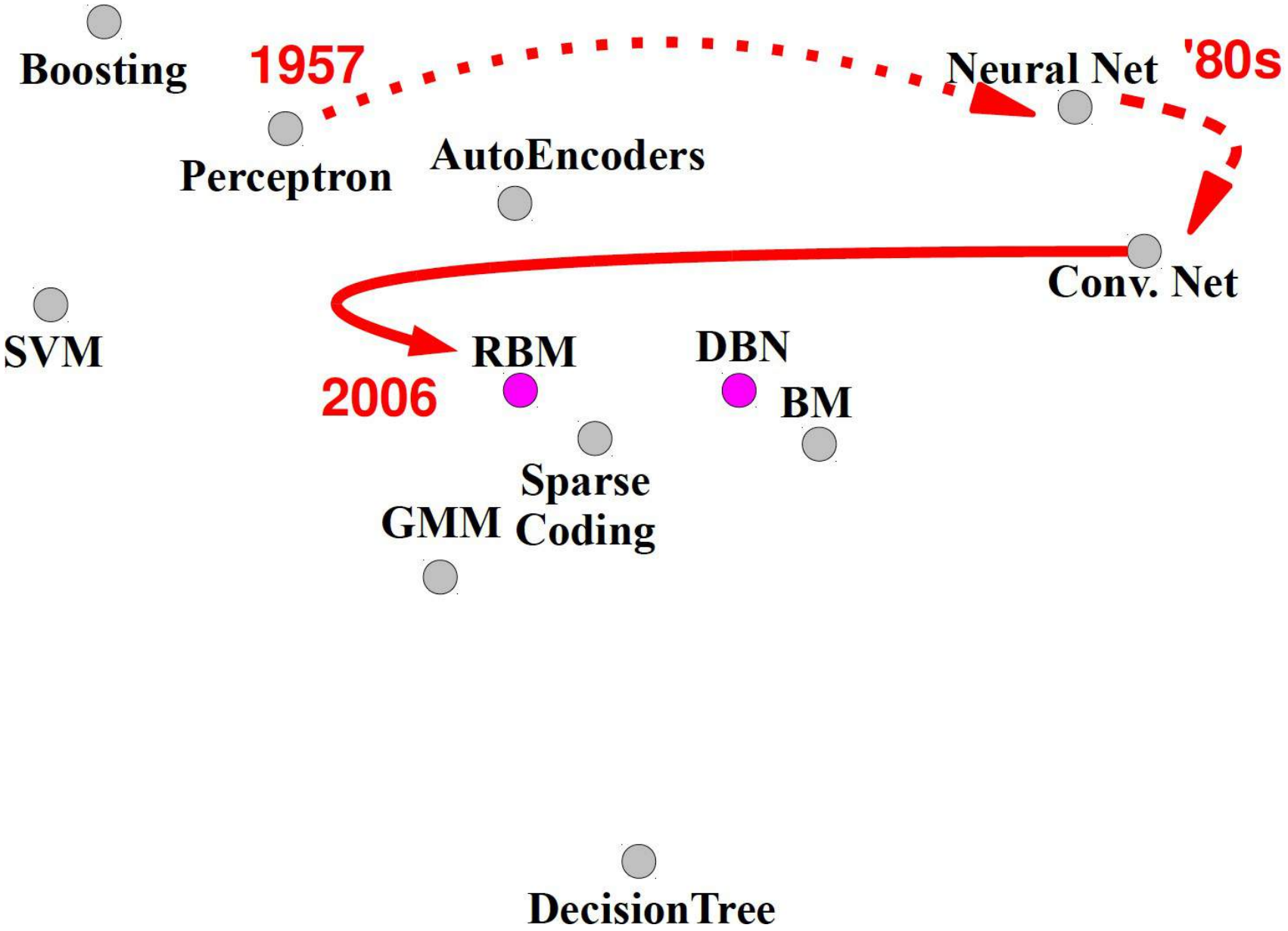
1957

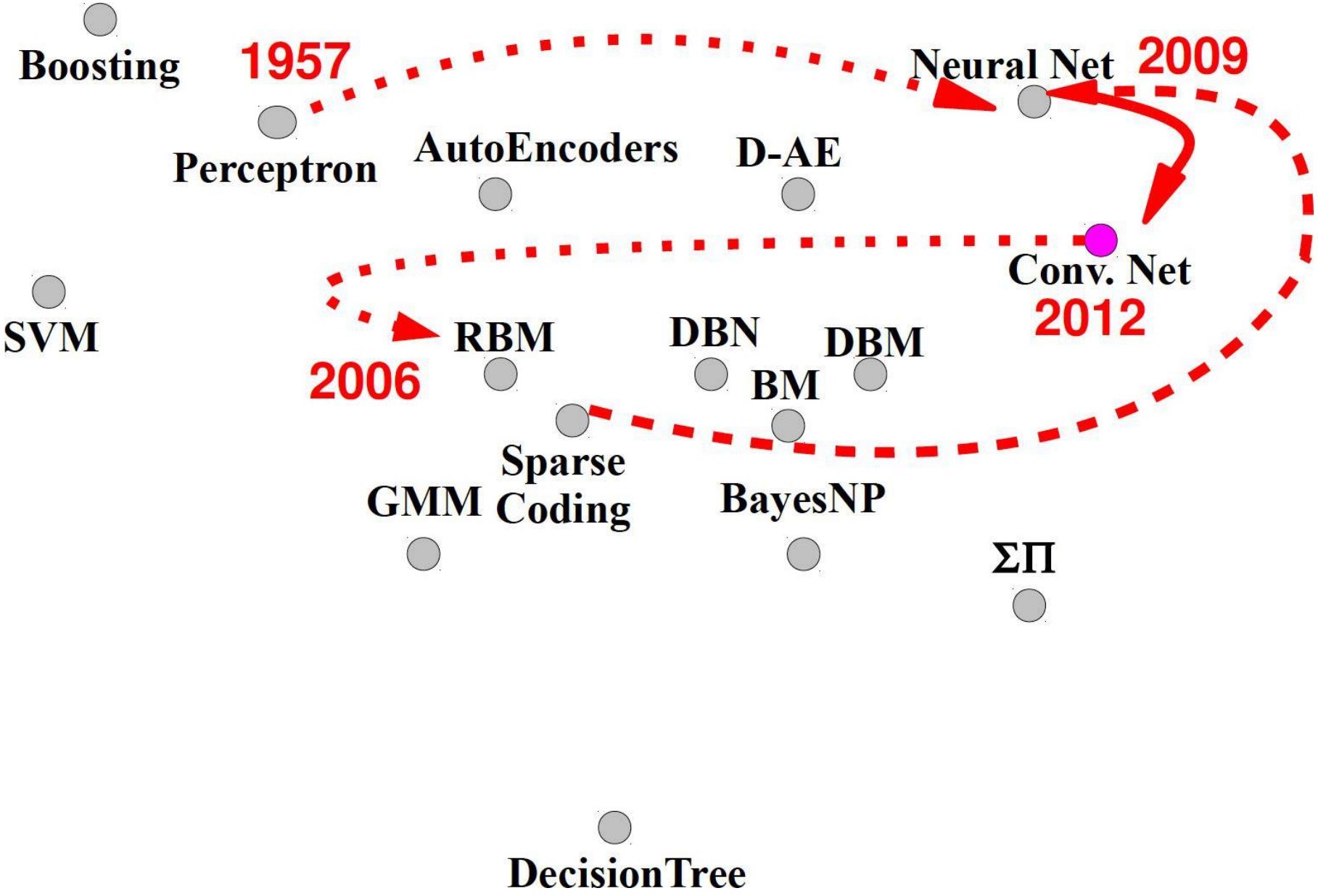
Perceptron

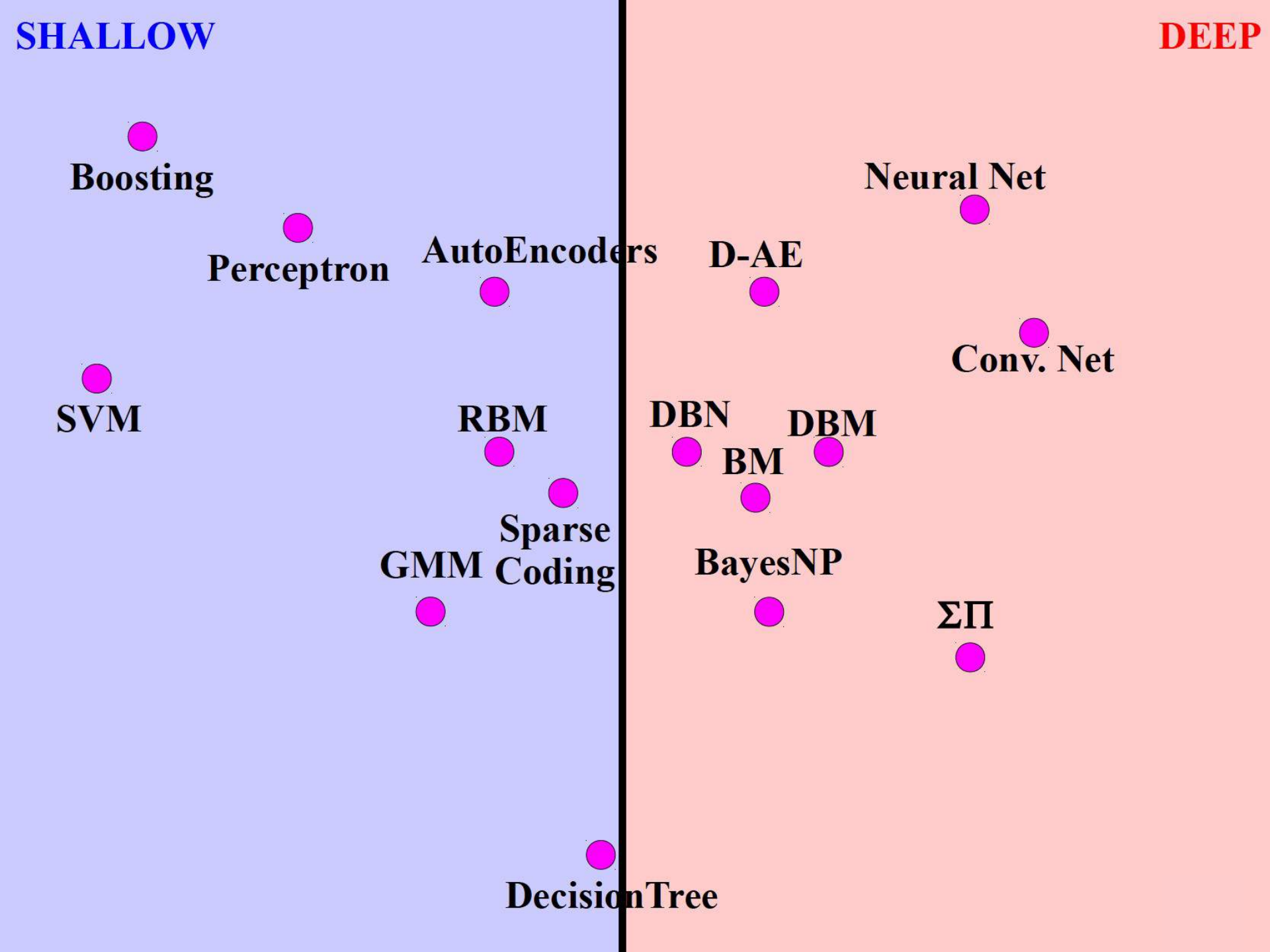
THE SPACE OF MACHINE LEARNING METHODS

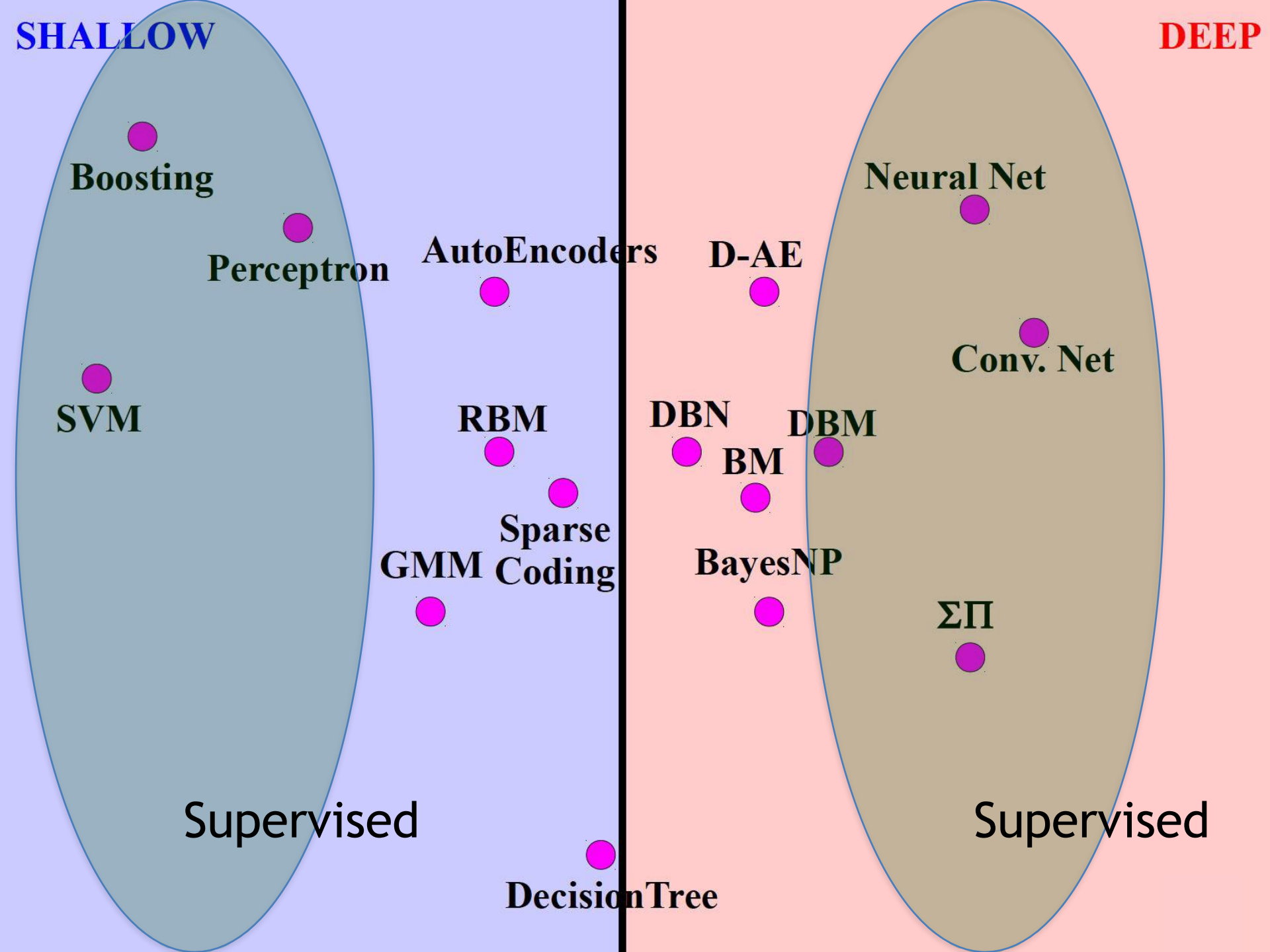

1957
Neural Net
'80s
Perceptron
AutoEncoders
BM


Boosting
1957
Neural Net
'80s
Perceptron
AutoEncoders
Conv. Net
SVM
BM
Sparse
Coding
GMM
DecisionTree
'90s – early '00s


Boosting
1957
Neural Net
'80s
Perceptron
AutoEncoders
Conv. Net
SVM
RBM
DBN
2006
BM
Sparse
GMM
Coding
DecisionTree

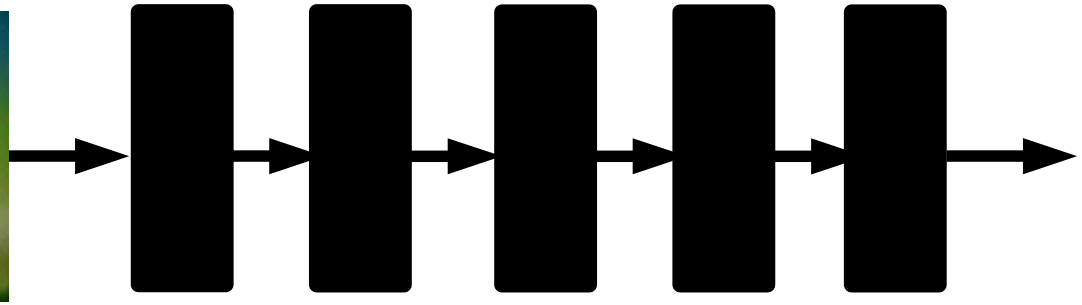

Boosting
1957
Neural Net
2009
Perceptron
AutoEncoders
D-AE
Conv. Net
2012
SVM
RBM
DBN
DBM
2006
BM
Sparse
Coding
GMM
BayesNP
ΣΠ
DecisionTree


SHALLOW
DEEP
Boosting
Perceptron
AutoEncoders
D-AE
Neural Net
Conv. Net
SVM
RBM
DBN
DBM
BM
Sparse Coding
GMM
BayesNP
ΣΠ
DecisionTree


SHALLOW
DEEP
Boosting
Perceptron
AutoEncoders
D-AE
Neural Net
Conv. Net
SVM
RBM
DBN
DBM
BM
Sparse Coding
GMM
BayesNP
ΣΠ
Supervised
Supervised
DecisionTree

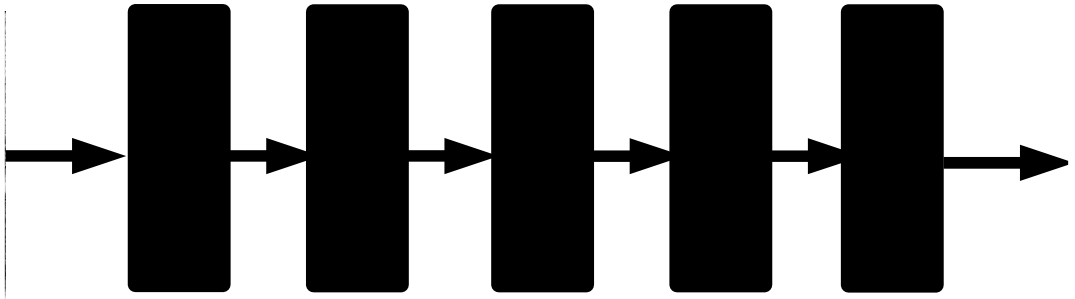
Supervised Deep Learning

Image Classification

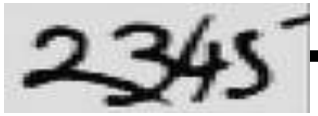


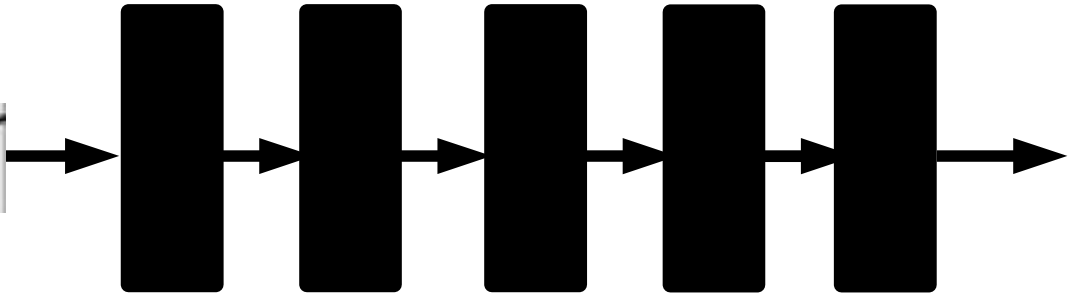
"dog"

Pose Estimation



OCR





"2 3 4 5"

Ranzato

Neural Networks

Neural Networks

Assumptions (for the next few slides):

- The input image is vectorized (disregard the spatial layout of pixels)
- The target label is discrete (classification)

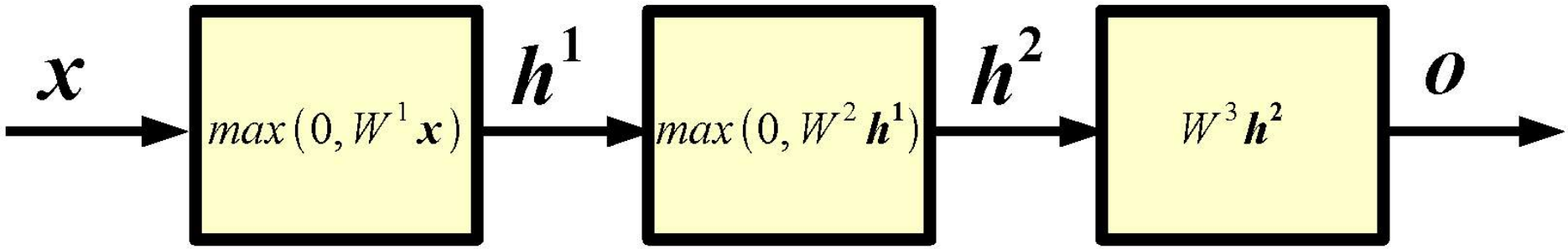
Question: what class of functions shall we consider to map the input into the output?

Answer: composition of simpler functions.

Follow-up questions: Why not a linear combination? What are the "simpler" functions? What is the interpretation?

Answer: later...

Neural Networks: example



x input

h^1 1-st layer hidden units

h^2 2-nd layer hidden units

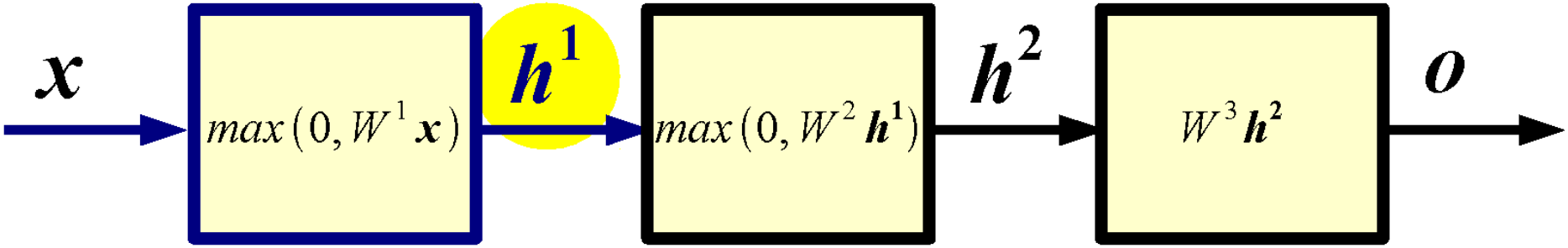
o output

Example of a 2 hidden layer neural network (or 4 layer network, counting also input and output).

Forward Propagation

Def.: Forward propagation is the process of computing the output of the network given its input.

Forward Propagation



$$\boldsymbol{x} \in R^{D} \qquad W^{1} \in R^{N_1 \times D} \qquad \boldsymbol{b}^{1} \in R^{N_1} \qquad \boldsymbol{h}^{1} \in R^{N_1}$$

$$\boldsymbol{h}^{1} = max(0, W^{1}\boldsymbol{x} + \boldsymbol{b}^{1})$$

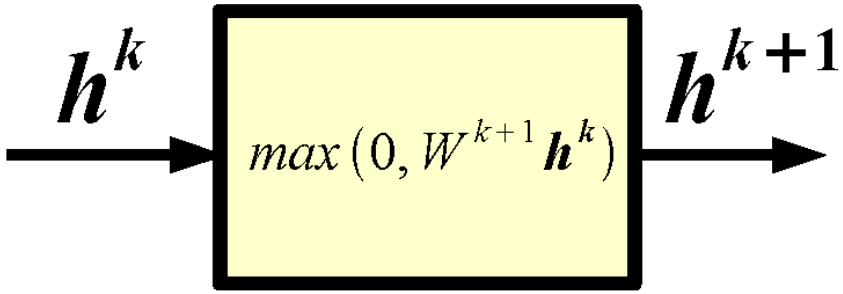
W^1 1-st layer weight matrix or weights

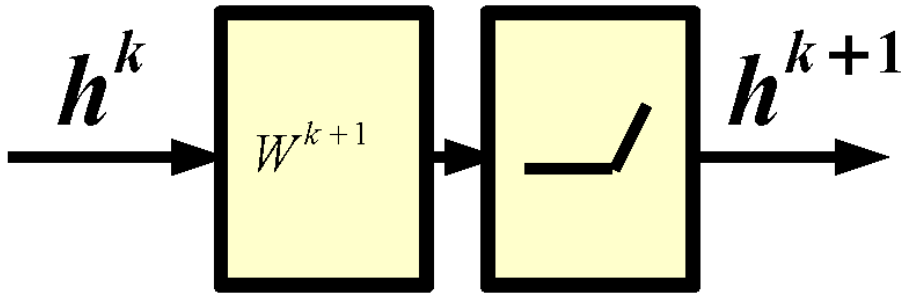
$\boldsymbol{b}^1$ 1-st layer biases

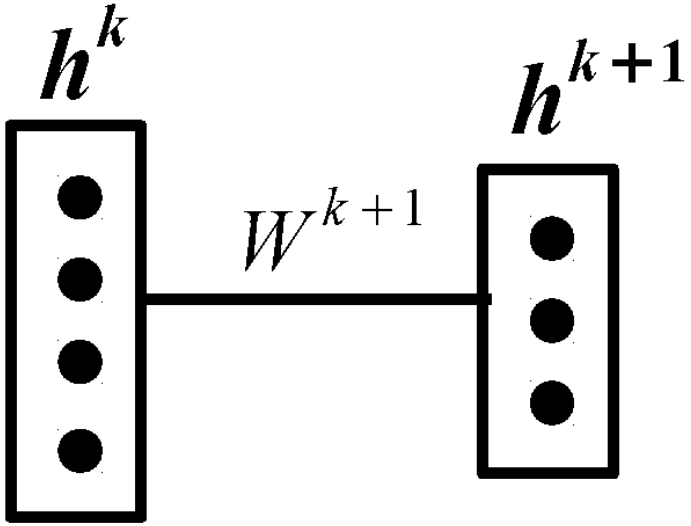
The non-linearity $u = max(0, v)$ is called **ReLU** in the DL literature.
Each output hidden unit takes as input all the units at the previous layer: each such layer is called "**fully connected**".

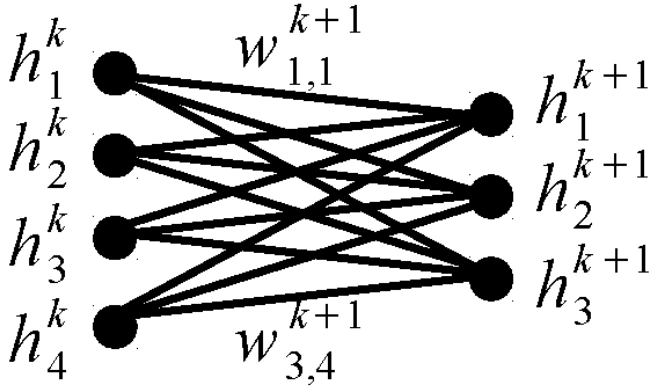
Ranzato

Alternative Graphical Representation



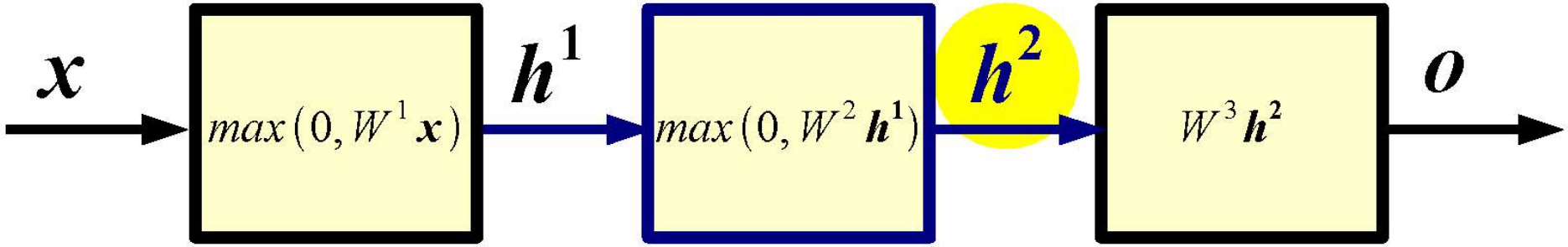






Ranzato

Forward Propagation



$$\mathbf{h^1} \in R^{N_1} \quad W^2 \in R^{N_2 \times N_1} \quad \mathbf{b^2} \in R^{N_2} \qquad \mathbf{h^2} \in R^{N_2}$$

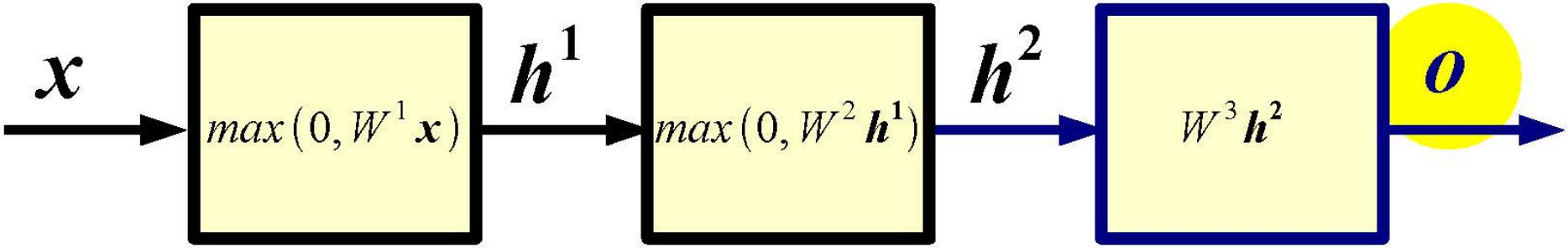
$$\mathbf{h^2} = max\left(0, W^2 \mathbf{h^1} + \mathbf{b^2}\right)$$

W^2 2-nd layer weight matrix or weights

$\mathbf{b^2}$ 2-nd layer biases

Ranzato

Forward Propagation



$\boldsymbol{h^2} \in R^{N_2}$ $W^3 \in R^{N_3 \times N_2}$ $\boldsymbol{b^3} \in R^{N_3}$ $\boldsymbol{o} \in R^{N_3}$

$$o = \sigma(W^3 \boldsymbol{h^2})$$

W^3 3-rd layer weight matrix or weights

$\boldsymbol{b^3}$ 3-rd layer biases

Ranzato

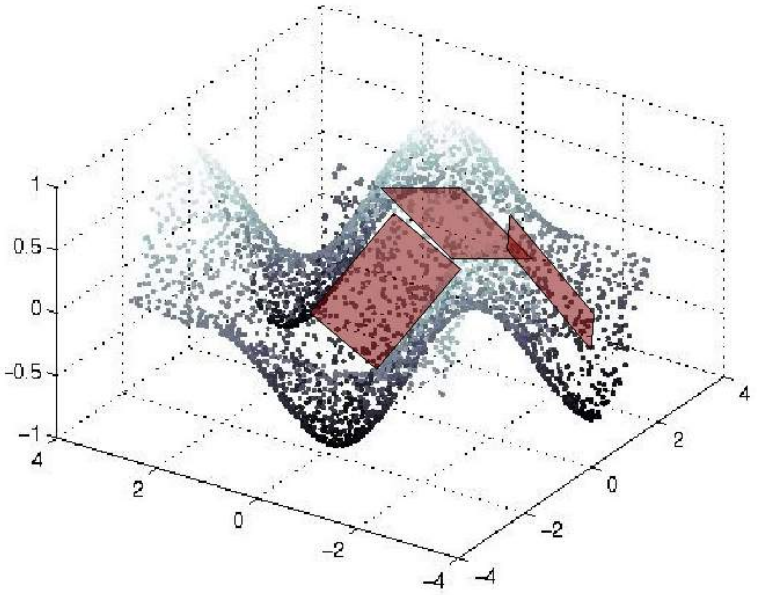
Interpretation

Question: Why can't the mapping between layers be linear?

Answer: Because composition of linear functions is a linear function. Neural network would reduce to (1 layer) logistic regression.

Question: What do ReLU layers accomplish?

Answer: Piece-wise linear tiling: mapping is locally linear.



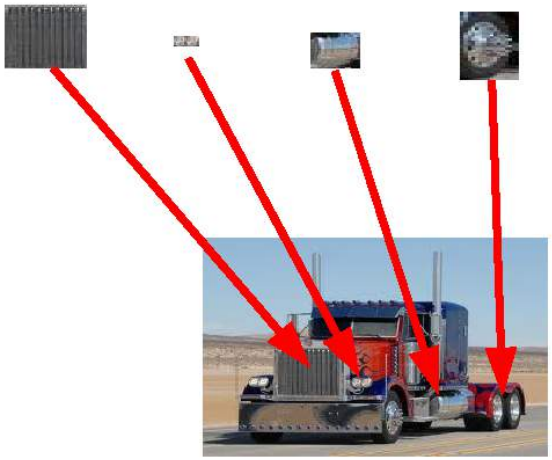
Ranzato

Interpretation

Question: Why do we need many layers?

Answer: When input has hierarchical structure, the use of a hierarchical architecture is potentially more efficient because intermediate computations can be re-used. DL architectures are efficient also because they use **distributed representations** which are shared across classes.

[0 0 **1** 0 0 0 0 **1** 0 0 **1** **1** 0 0 **1** 0 …] truck feature



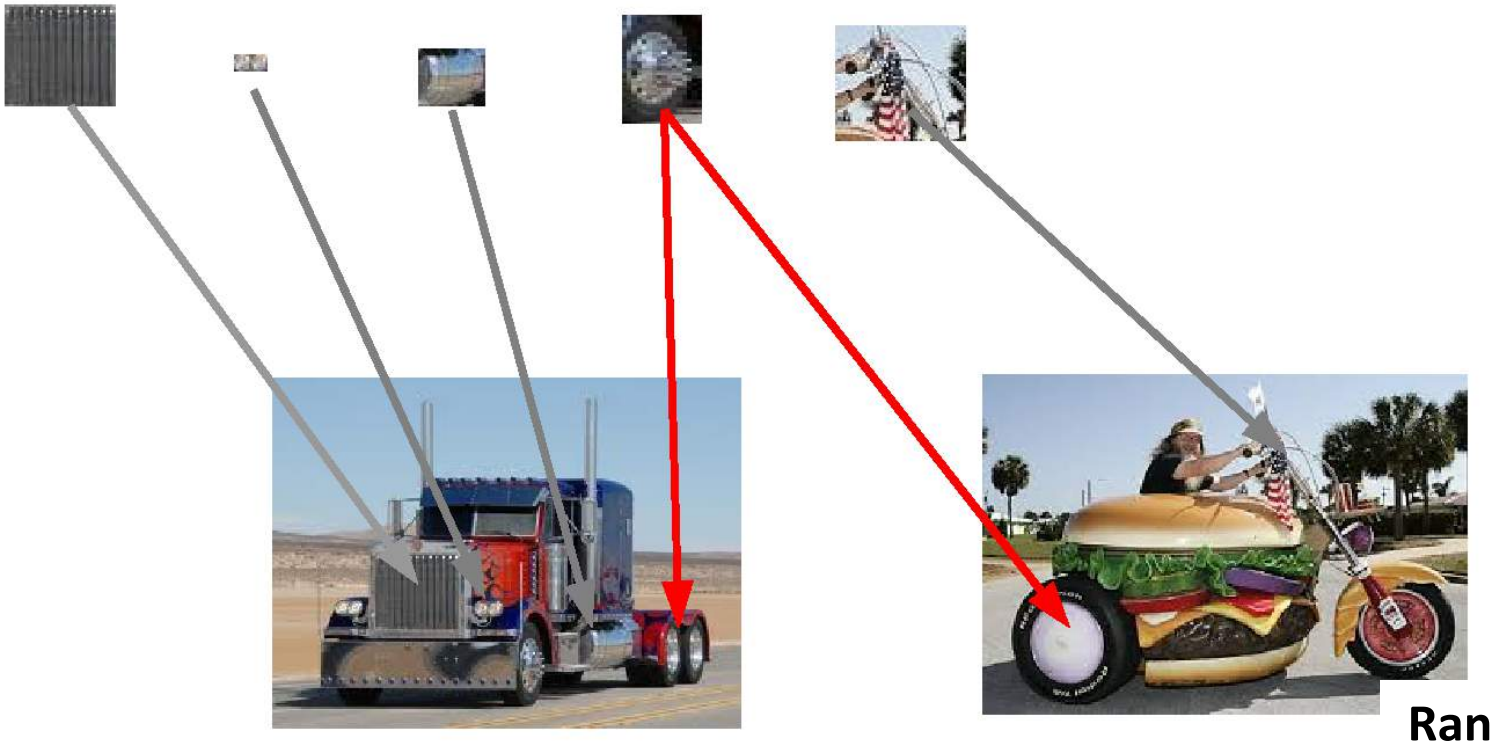
Exponentially more efficient than a 1-of-N representation (a la k-means)

Ranzato

Interpretation

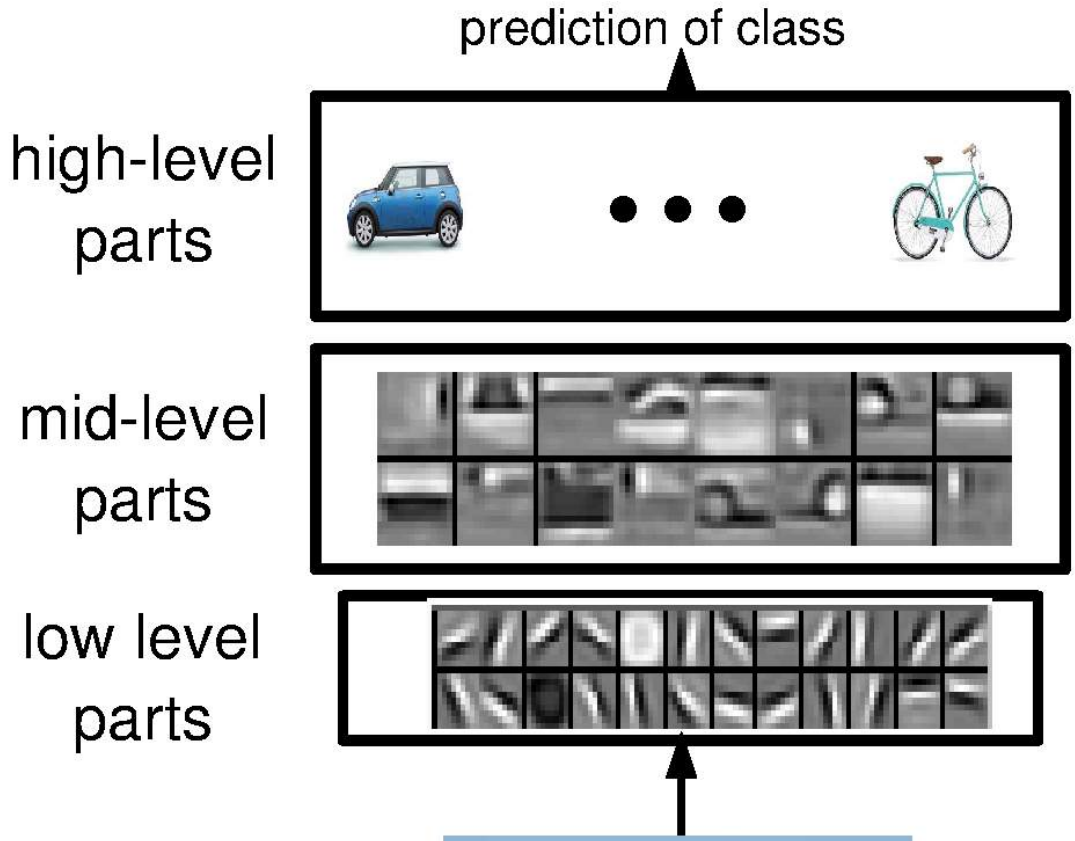
[1 1 0 0 0 1 0 **1** 0 0 0 0 1 1 0 1…] motorbike

[0 0 1 0 0 0 0 **1** 0 0 1 1 0 0 1 0 …] truck



Ranzato

Interpretation



- distributed representations
- feature sharing
- compositionality

Lee et al. “Convolutional DBN's ...” ICML 2009

Ranzato

Interpretation

Question: What does a hidden unit do?

Answer: It can be thought of as a classifier or feature detector.

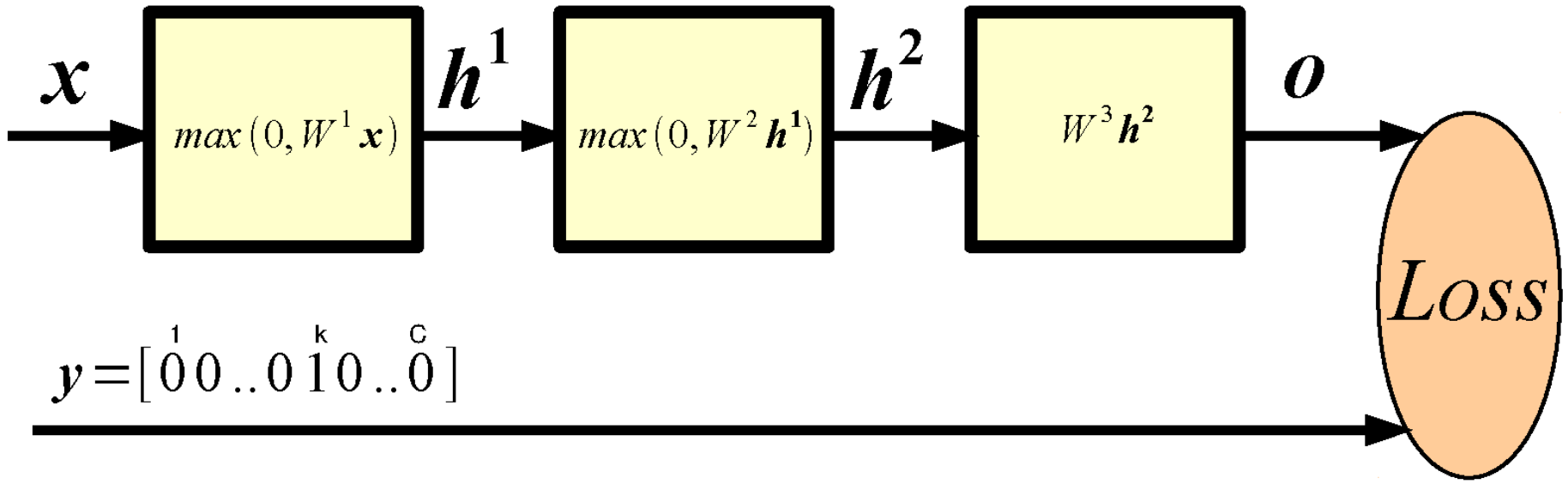
Question: How many layers? How many hidden units?

Answer: Cross-validation or hyper-parameter search methods are the answer. In general, the wider and the deeper the network the more complicated the mapping.

Question: How do I set the weight matrices?

Answer: Weight matrices and biases are learned.
First, we need to define a measure of quality of the current mapping.
Then, we need to define a procedure to adjust the parameters.

How Good is a Network?



Probability of class k given input (softmax):

$$p(c_k = 1|\boldsymbol{x}) = \frac{e^{o_k}}{\sum_{j=1}^{C} e^{o_j}}$$

(Per-sample) **Loss**; e.g., negative log-likelihood (good for classification of small number of classes):

$$L(\boldsymbol{x}, y; \boldsymbol{\theta}) = -\sum_j y_j \log p(c_j|\boldsymbol{x})$$

Ranzato

Training

Learning consists of minimizing the loss (plus some regularization term) w.r.t. parameters over the whole training set.

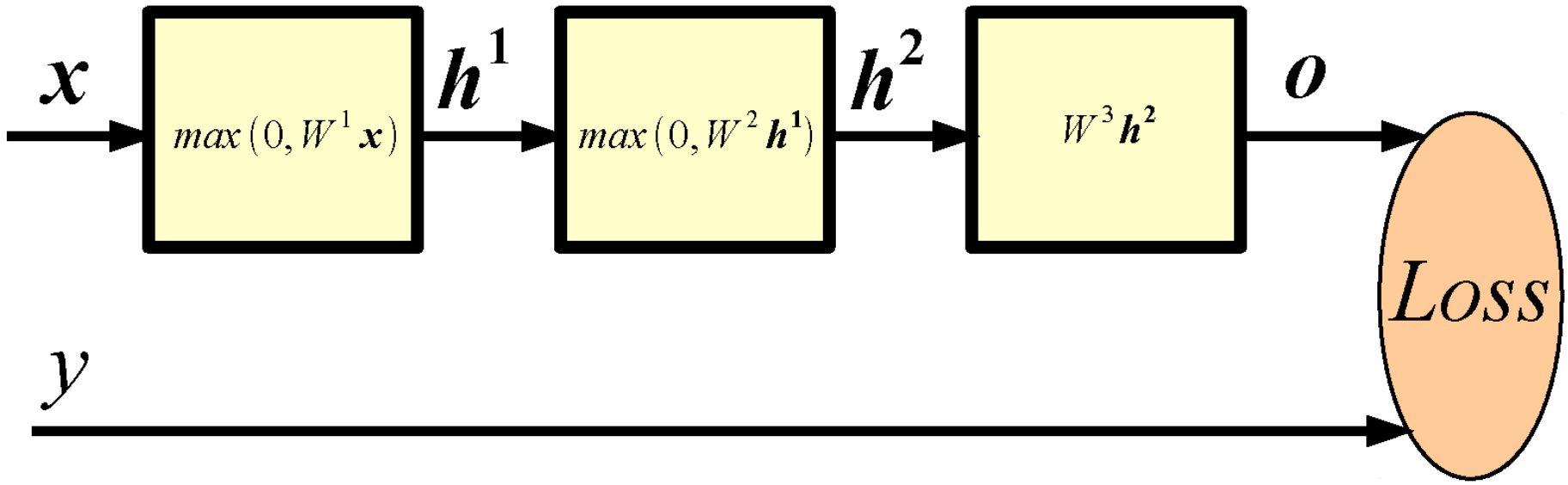
$$\theta^* = argmin_\theta \sum_{n=1}^{N} L(x_n, y_n;\ \theta)$$

Question: How to minimize a complicated function of the parameters?

Answer: Chain rule, a.k.a. **Backpropagation**! That is the procedure to compute gradients of the loss w.r.t. parameters in a multi-layer neural network.

Rumelhart et al. "Learning internal representations by back-propagating.." Nature 1986

Key Idea: Wiggle To Decrease Loss



Let's say we want to decrease the loss by adjusting $W^1_{i,j}$.
We could consider a very small $\epsilon = 1\text{e-}6$ and compute:

$$L(\boldsymbol{x}, y; \boldsymbol{\theta})$$

$$L(\boldsymbol{x}, y; \boldsymbol{\theta} \setminus W^1_{i,j}, W^1_{i,j} + \epsilon)$$

Then, update:

$$W^1_{i,j} \leftarrow W^1_{i,j} + \epsilon \, sgn(L(\boldsymbol{x}, y; \boldsymbol{\theta}) - L(\boldsymbol{x}, y; \boldsymbol{\theta} \setminus W^1_{i,j}, W^1_{i,j} + \epsilon))$$

Derivative w.r.t. Input of Softmax

$$p(c_k=1|\boldsymbol{x})=\frac{e^{o_k}}{\sum_j e^{o_j}}$$

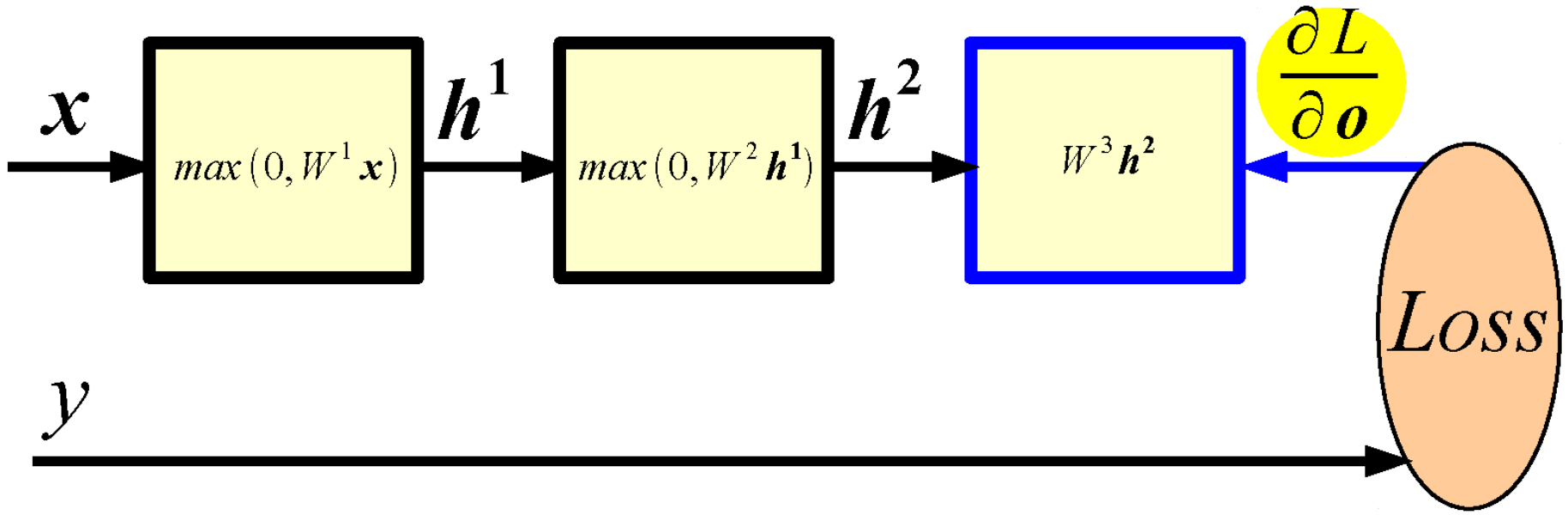
$$L(\boldsymbol{x}, y; \boldsymbol{\theta})=-\sum_j y_j \log p(c_j|\boldsymbol{x}) \qquad \boldsymbol{y}=[\overset{1}{0}0..0\overset{k}{1}0..\overset{C}{0}]$$

By substituting the fist formula in the second, and taking the derivative w.r.t. $\boldsymbol{o}$ we get:

$$\frac{\partial L}{\partial o} = p(c|\boldsymbol{x}) - \boldsymbol{y}$$

Ranzato

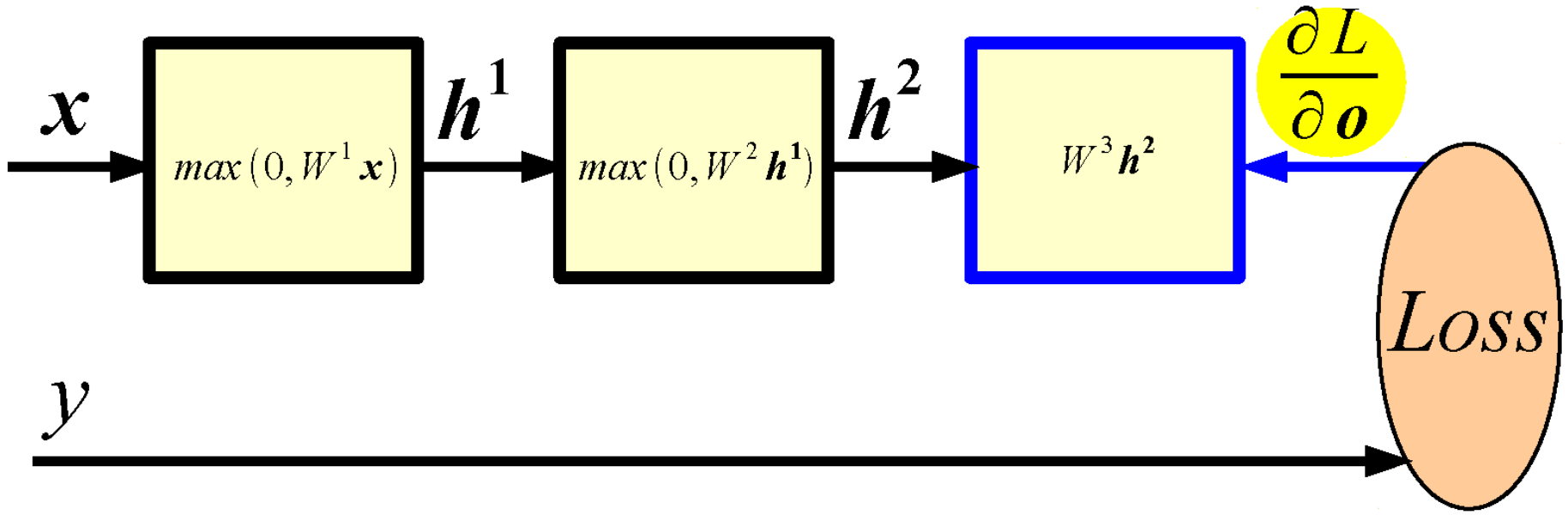
Backward Propagation



Given $\partial L / \partial \boldsymbol{o}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^2}$$

Backward Propagation

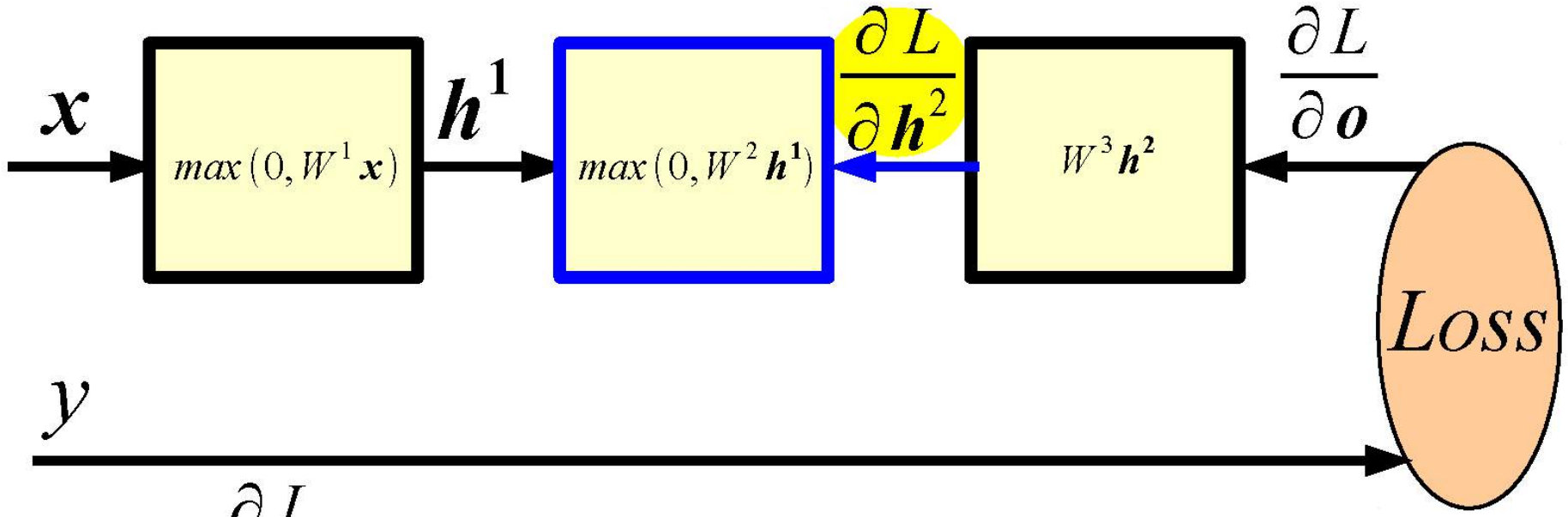


Given $\partial L / \partial \boldsymbol{o}$ and assuming we can easily compute the Jacobian of each module, we have:

$$\frac{\partial L}{\partial W^3} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial W^3} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = \frac{\partial L}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}^2}$$

$$\frac{\partial L}{\partial W^3} = (p(c|\boldsymbol{x}) - \boldsymbol{y})\ \boldsymbol{h}^{2\,T} \qquad \frac{\partial L}{\partial \boldsymbol{h}^2} = W^{3\,T}(p(c|\boldsymbol{x}) - \boldsymbol{y})$$

Backward Propagation

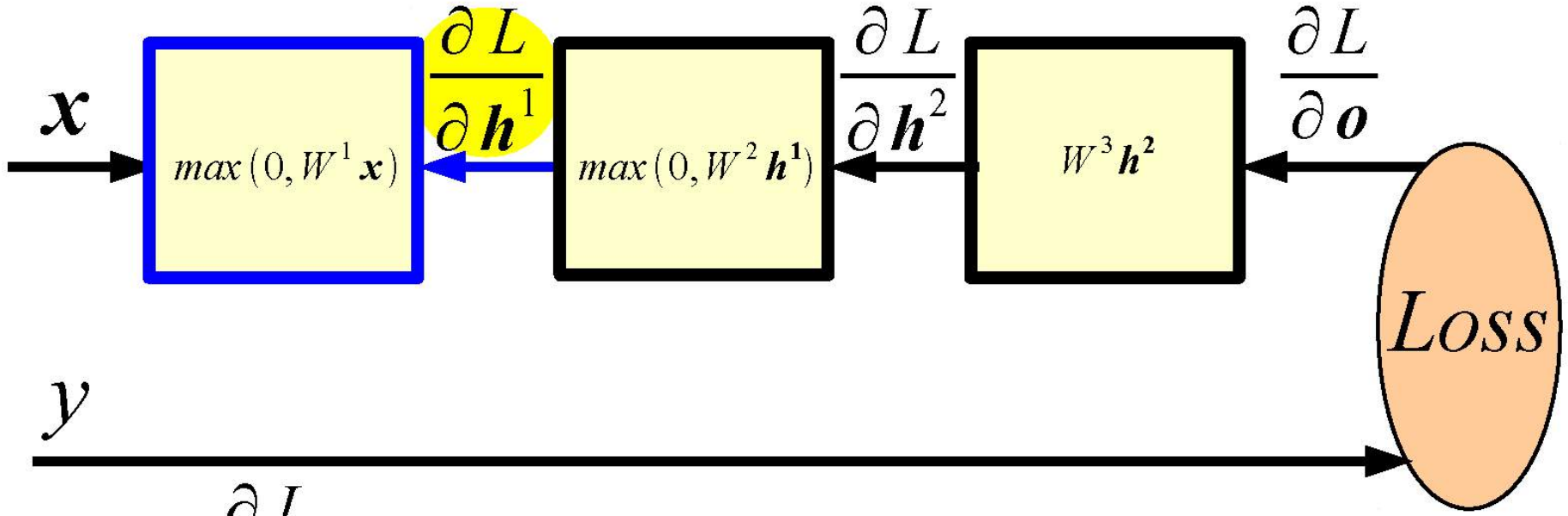


Given $\frac{\partial L}{\partial \boldsymbol{h}^2}$ we can compute now:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \boldsymbol{h}^2}\frac{\partial \boldsymbol{h}^2}{\partial W^2} \qquad \frac{\partial L}{\partial \boldsymbol{h}^1} = \frac{\partial L}{\partial \boldsymbol{h}^2}\frac{\partial \boldsymbol{h}^2}{\partial \boldsymbol{h}^1}$$

Ranzato

Backward Propagation

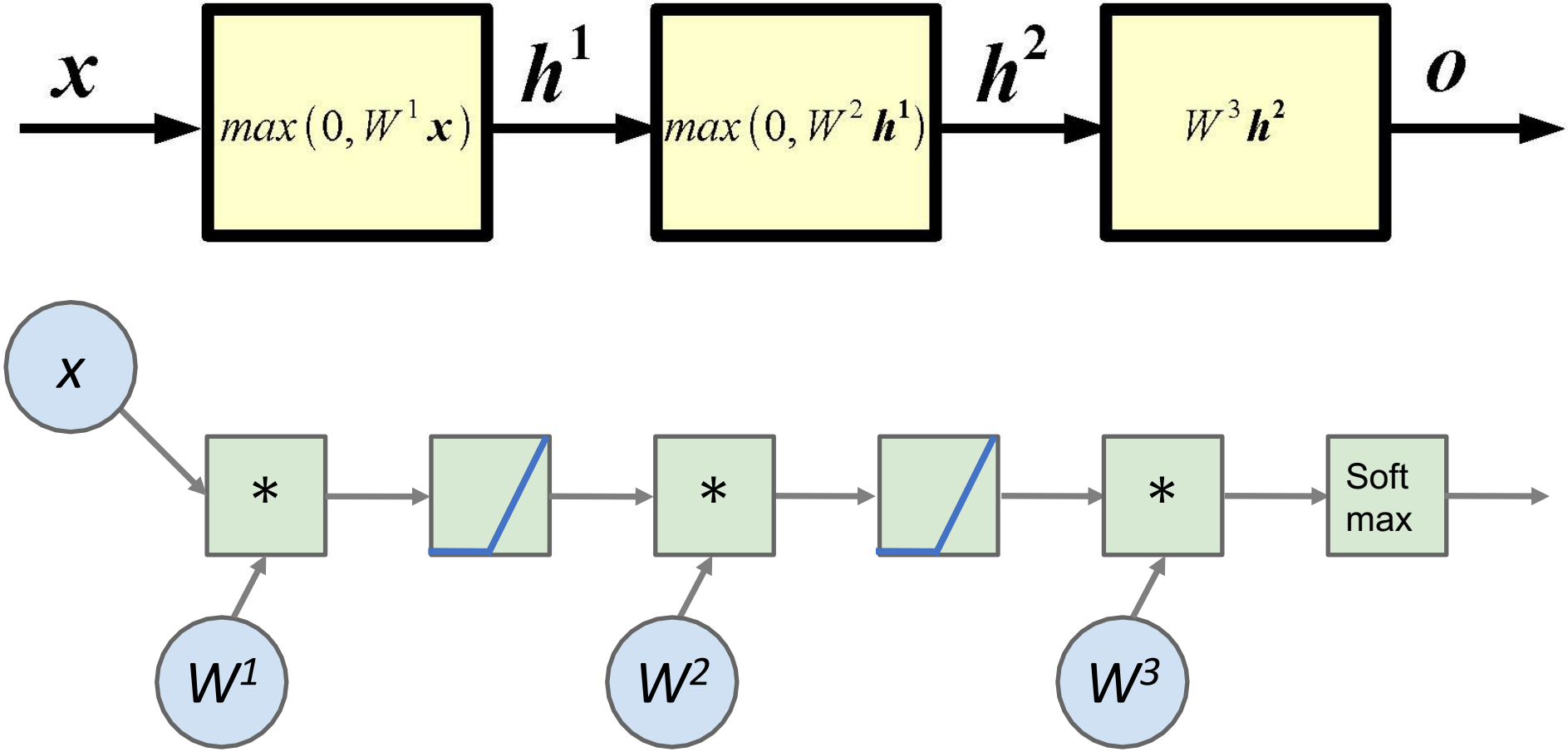


Given $\frac{\partial L}{\partial h^1}$ we can compute now:

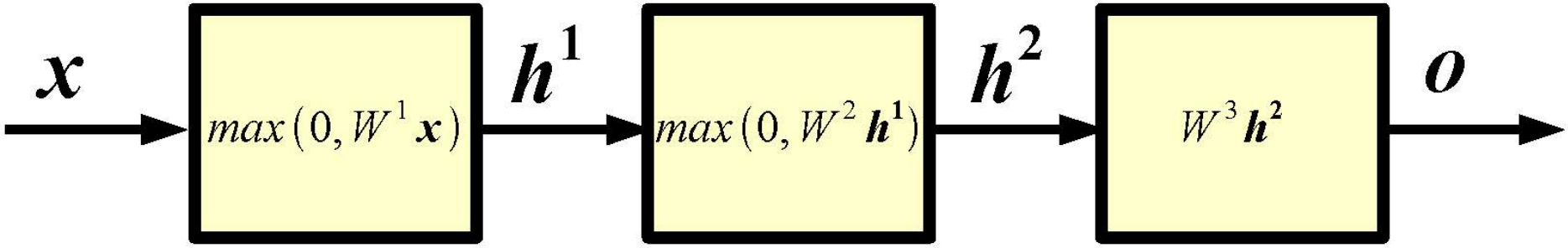
$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial h^1} \frac{\partial h^1}{\partial W^1}$$

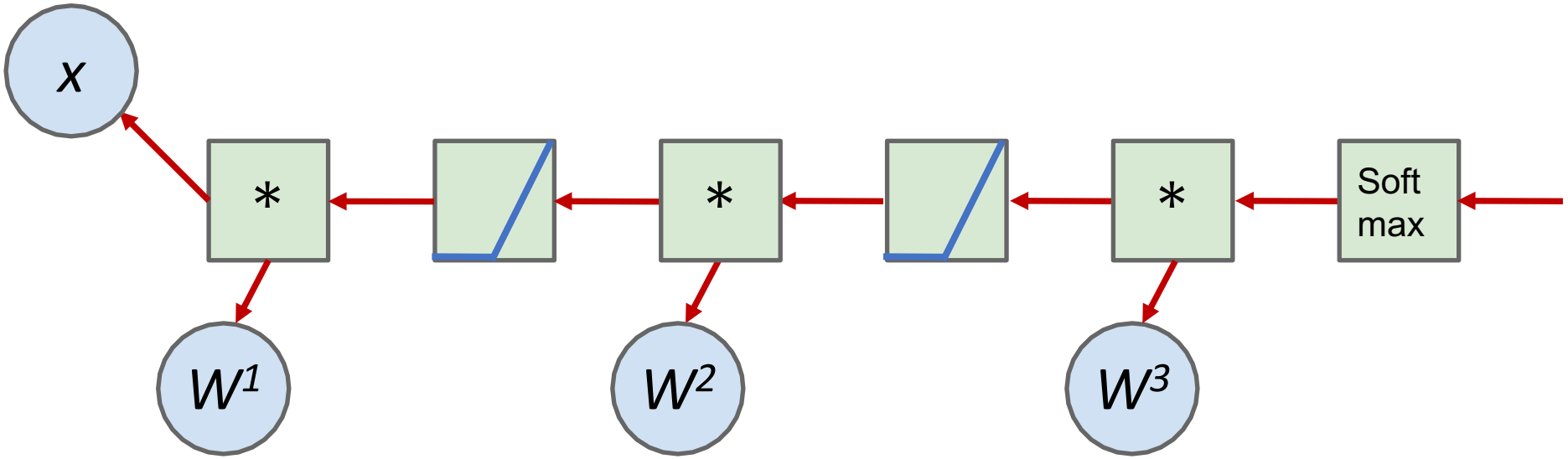
Ranzato

Backprop on Computational Graph

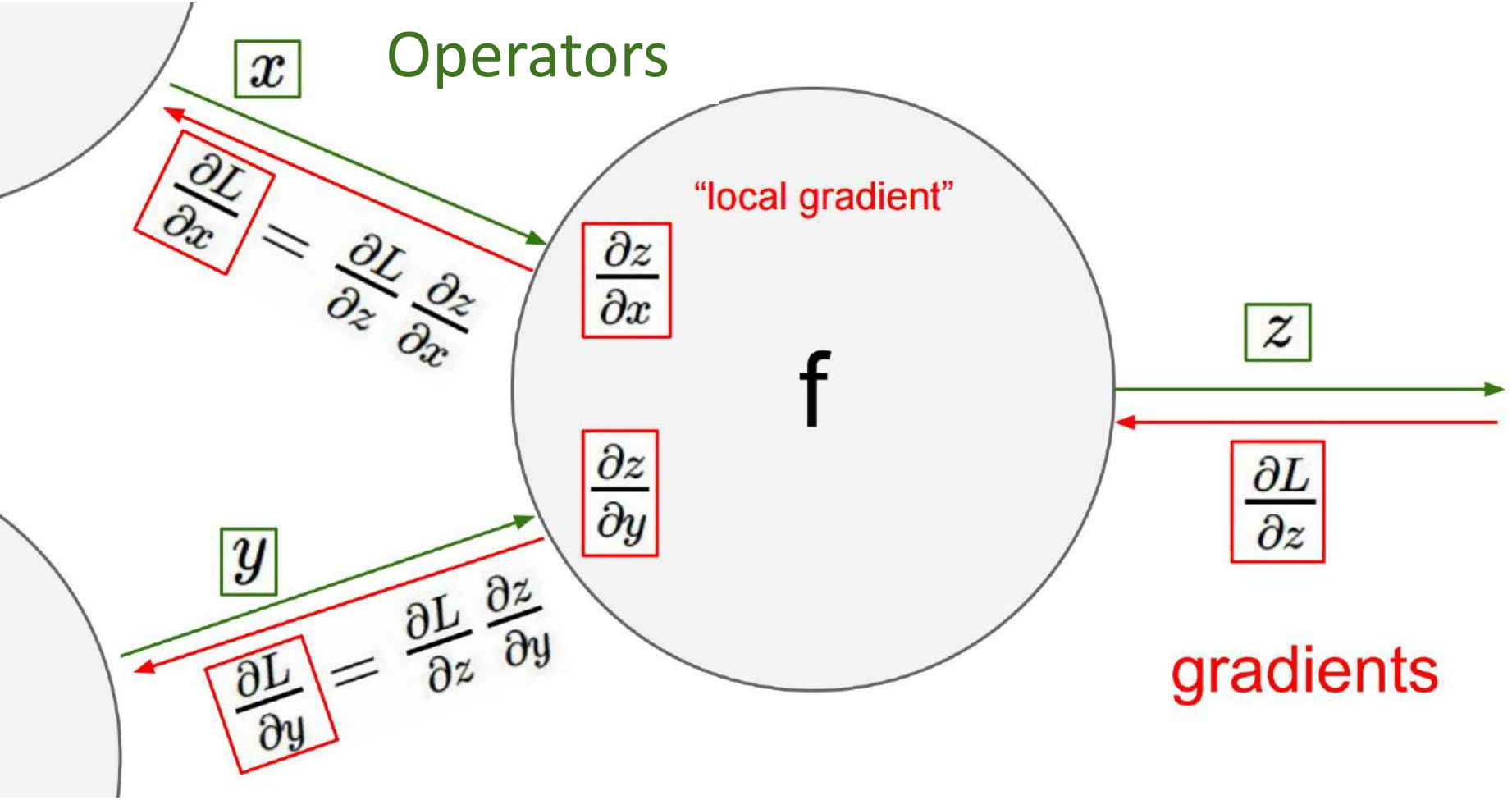


Backprop on Computational Graph





Understanding Backprop



Backward Propagation

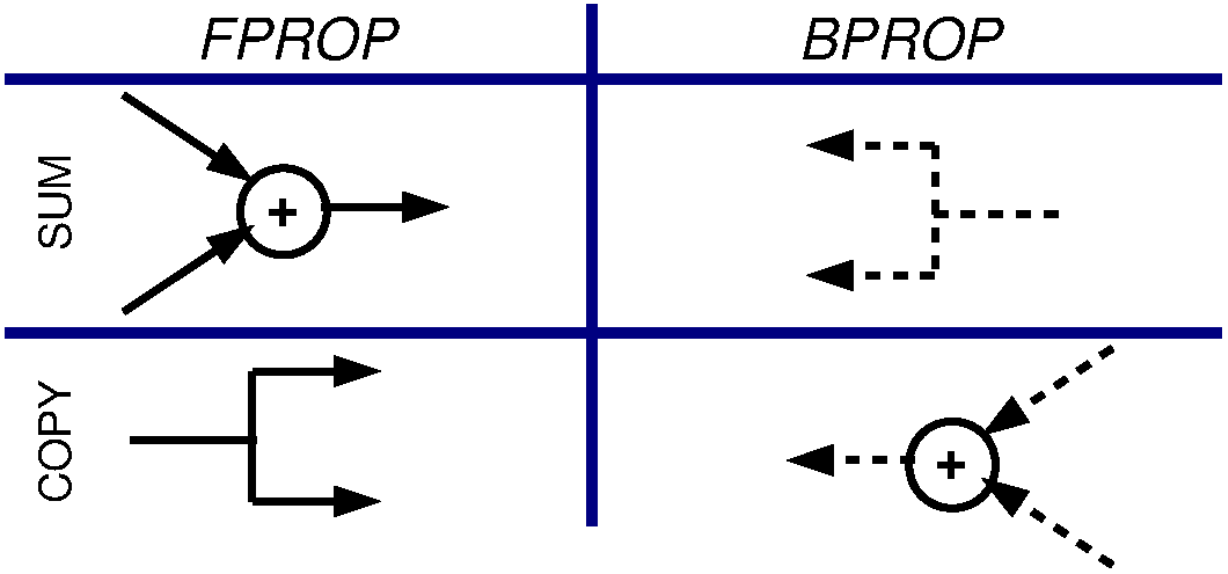
Question: Does BPROP work with ReLU layers only?

Answer: Nope, any a.e. differentiable transformation works.

Question: What's the computational cost of BPROP?

Answer: About twice FPROP (need to compute gradients w.r.t. input and parameters at every layer).

Note: FPROP and BPROP are dual of each other. E.g.,:



Ranzato

Optimization

Stochastic Gradient Descent (on mini-batches):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}, \eta \in (0,1)$$ Learning Rate

Stochastic Gradient Descent with Momentum:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\Delta}$$

$$\boldsymbol{\Delta} \leftarrow 0.9 \boldsymbol{\Delta} + \frac{\partial L}{\partial \boldsymbol{\theta}}$$

Note: there are many other variants...

Ranzato

Optimization

Loop:

- **Sample** a batch of data
- **Forward** prop it through the graph (network), get loss
- **Backprop** to calculate the gradients
- **Update** the parameters using the gradient

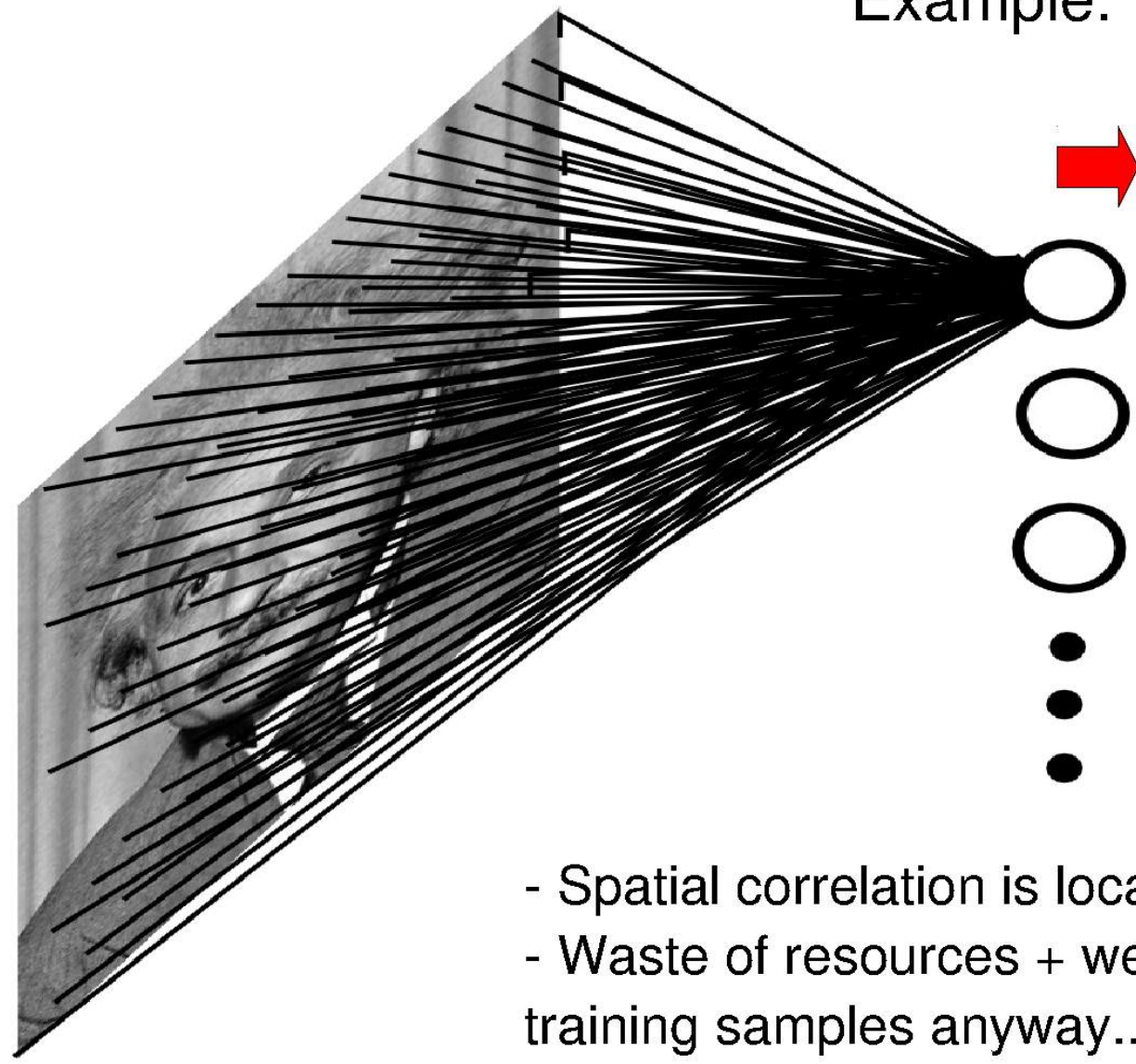
Till convergence?

Building Blocks of (Convolutional) Networks

- Fully connect layer (linear operation)
- Convolutional layer (still linear)
- Nonlinear activation functions
- Pooling layer (reduce the feature resolution)
- Normalization layers
  - local contrast normalization
  - batch normalization
- Output normalization layers (sigmoid, softmax)
- Loss functions
  - Cross entropy
  - L1/2 norm
  - KL divergence

All operations must be differentiable?

Fully Connected Layer



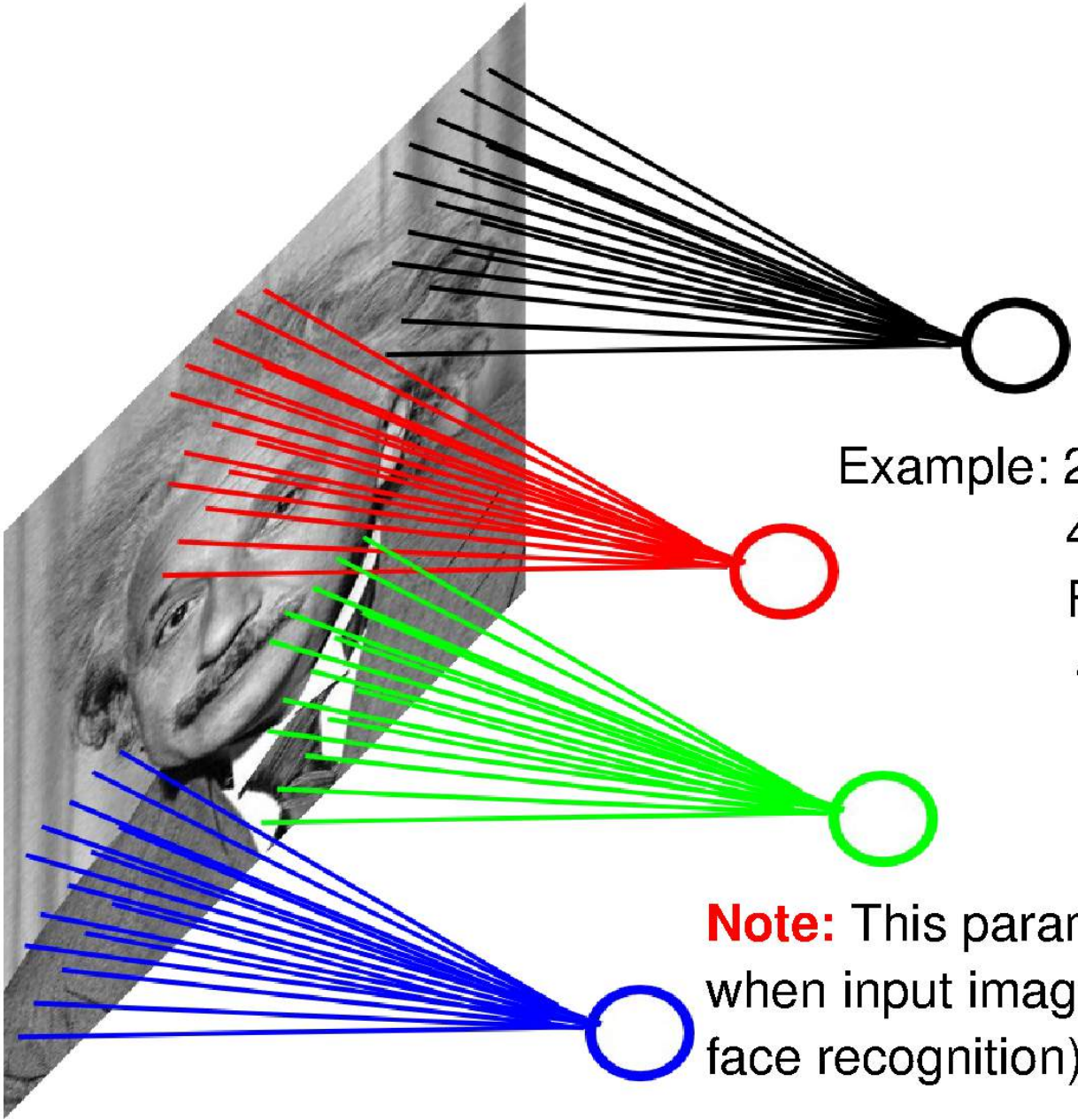
Example: 200x200 image

40K hidden units

~2B parameters!!!

- Spatial correlation is local
- Waste of resources + we have not enough training samples anyway..

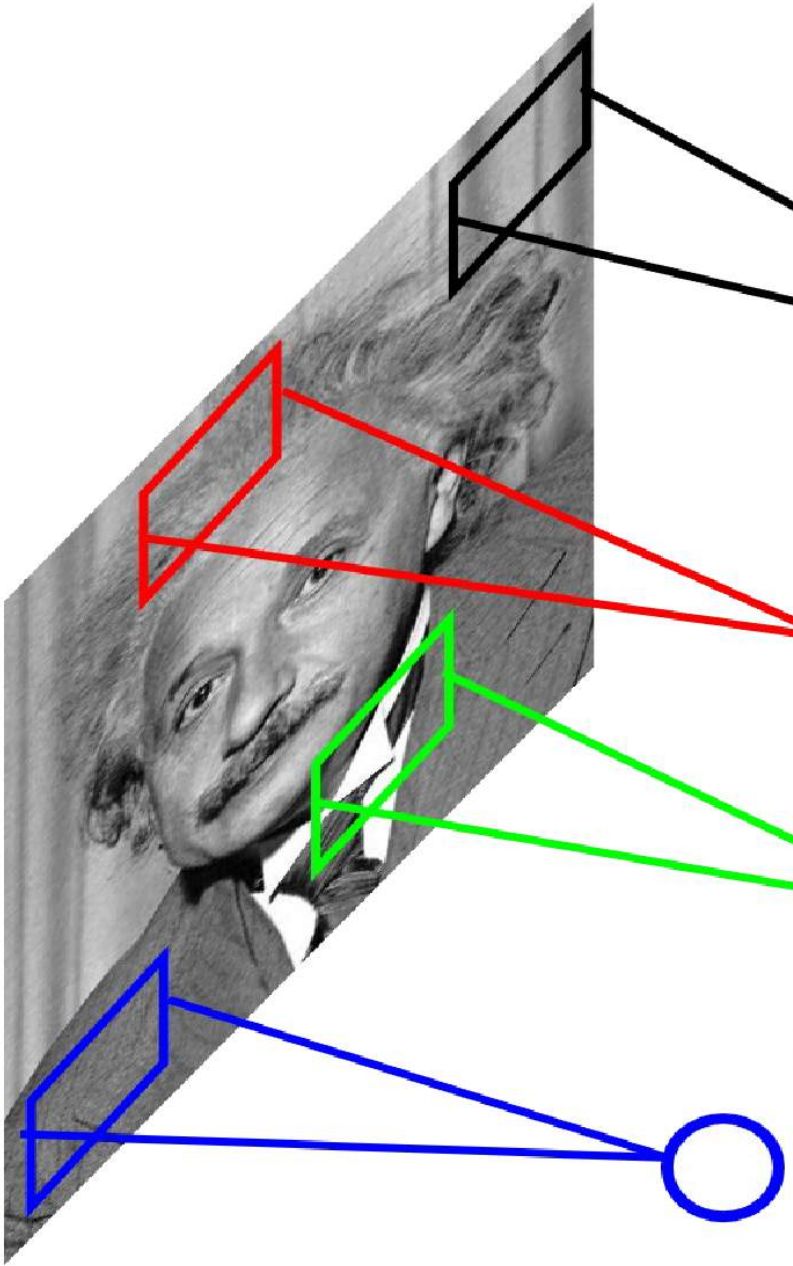
Locally Connected Layer



Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

Locally Connected Layer

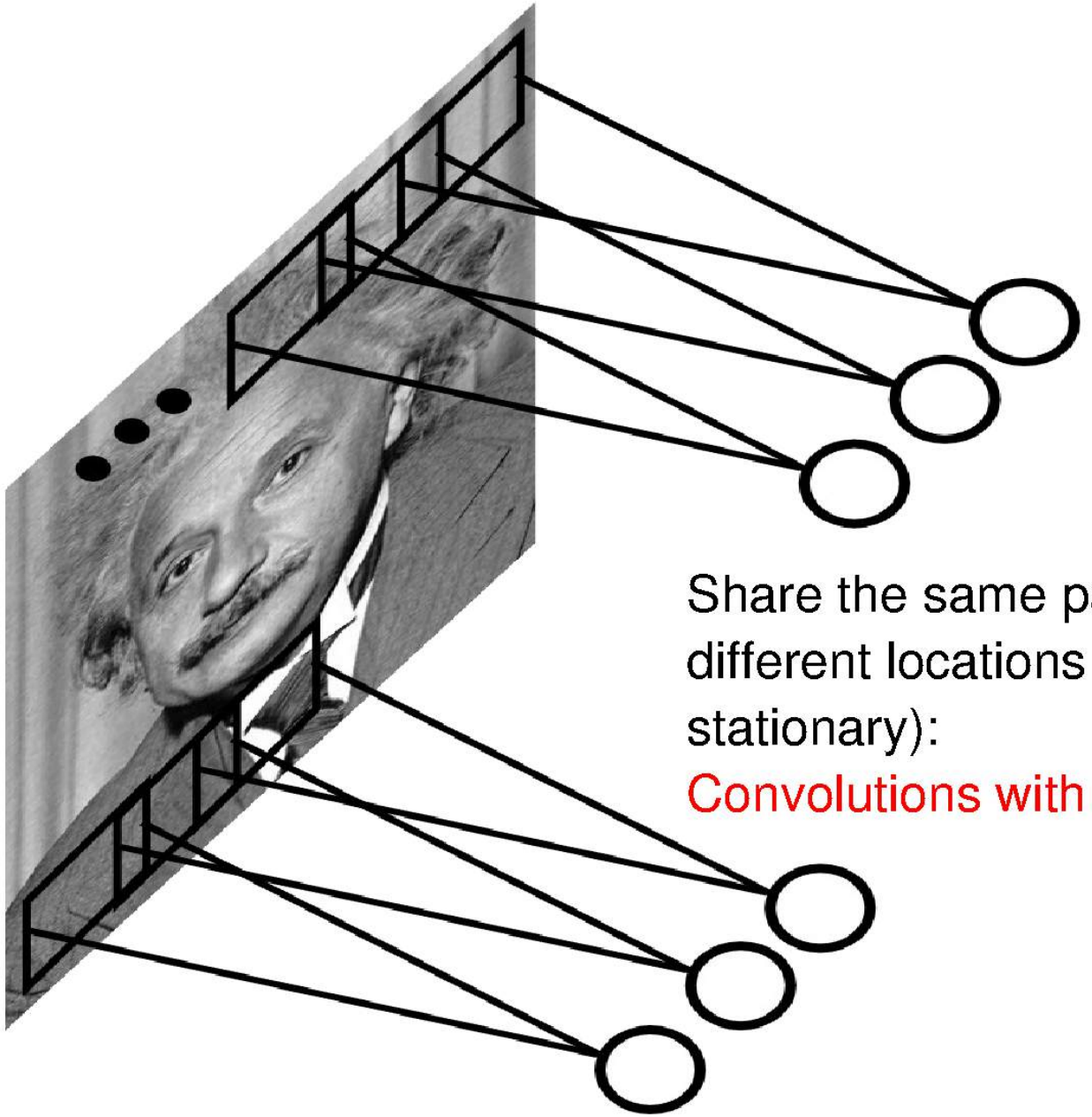


STATIONARITY? Statistics is similar at different locations

Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).

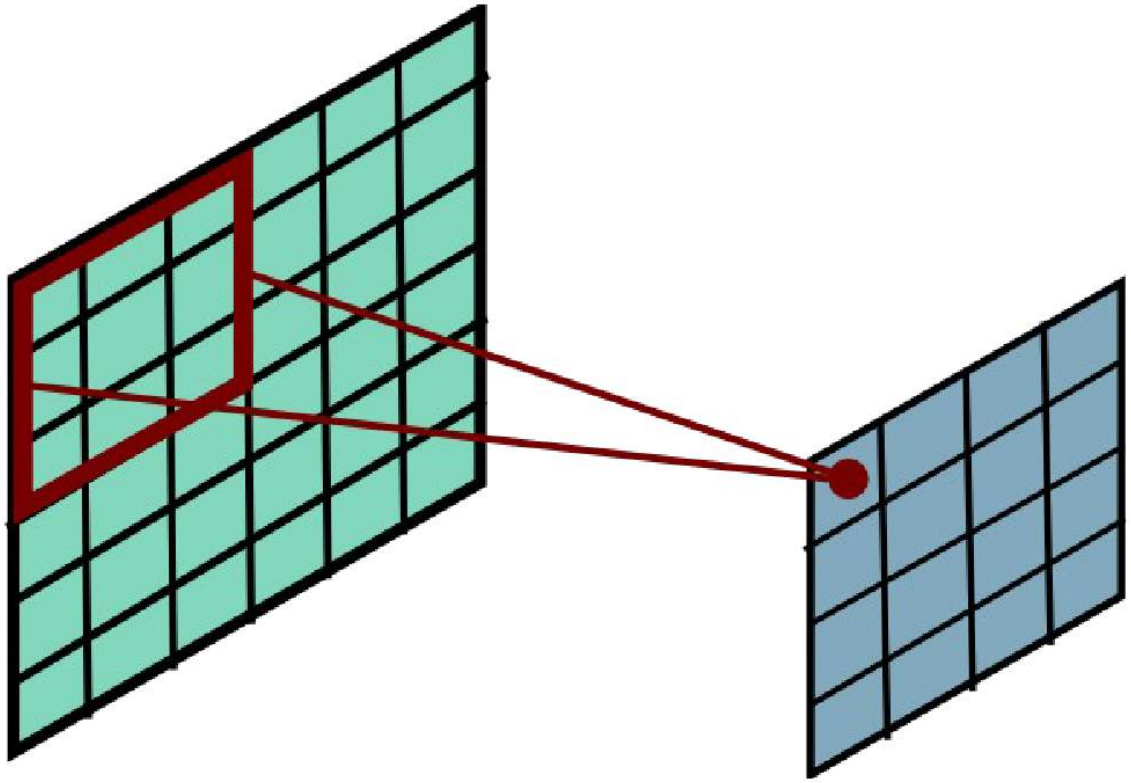
Convolutional Layer



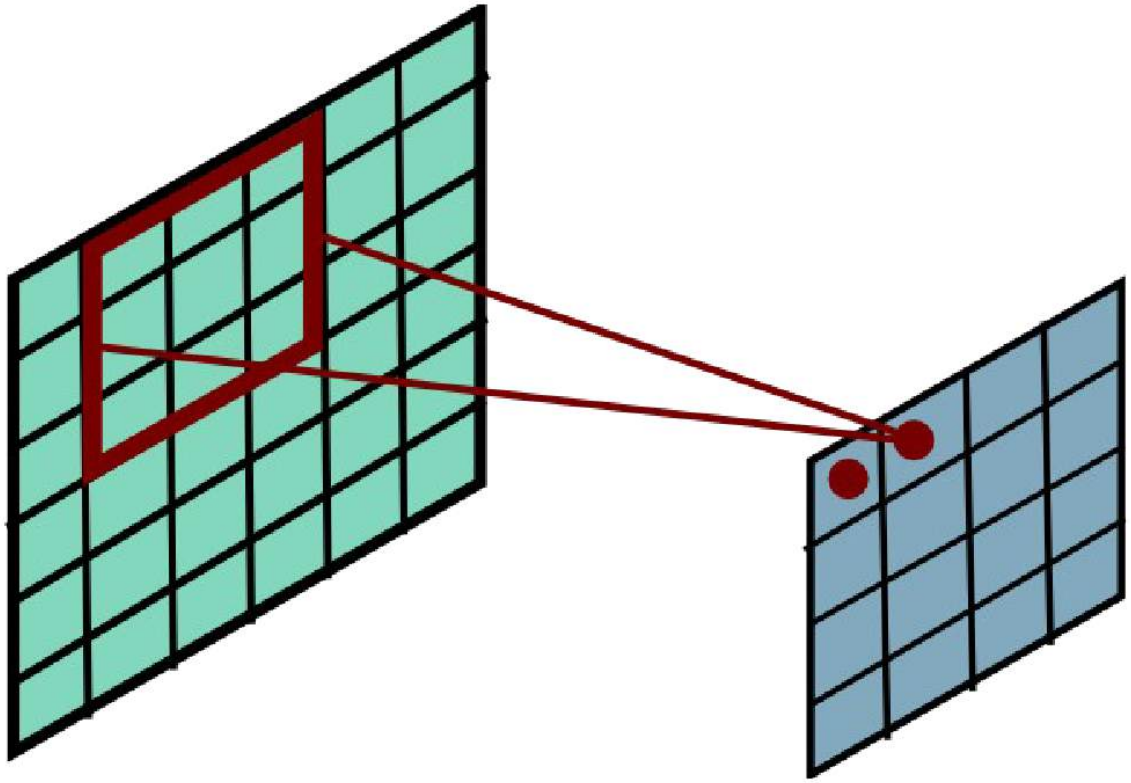
Share the same parameters across different locations (assuming input is stationary):

Convolutions with learned kernels

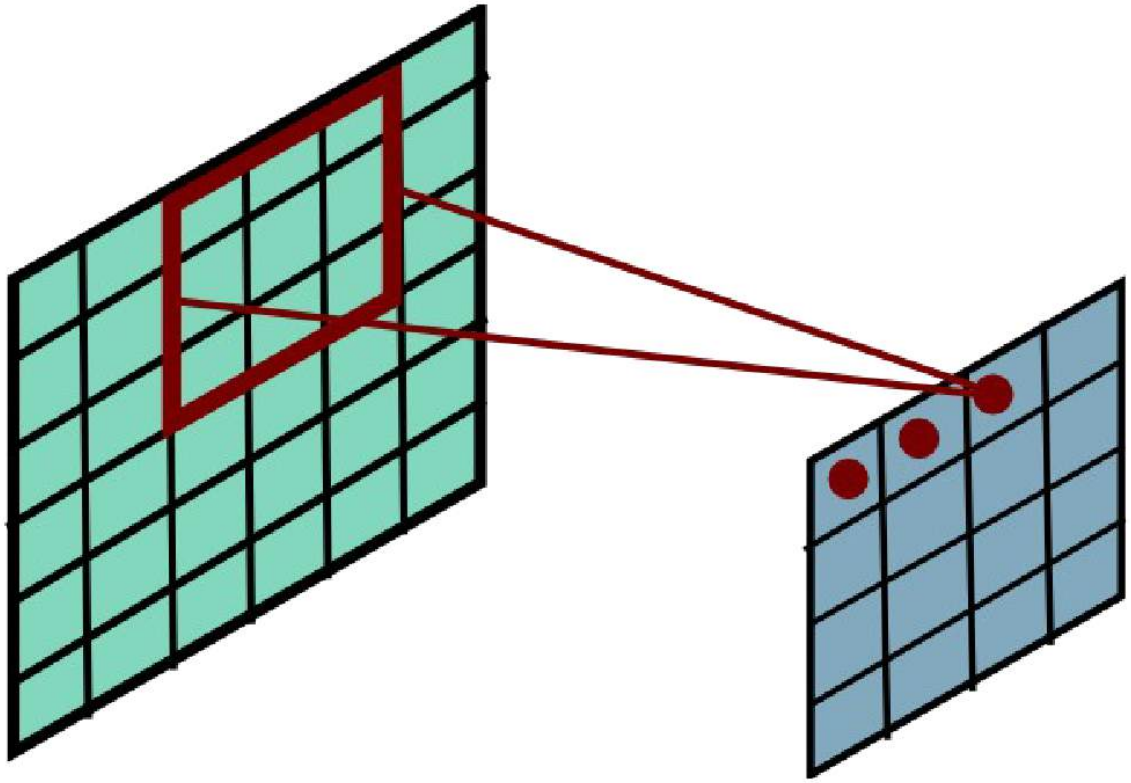
Convolutional Layer



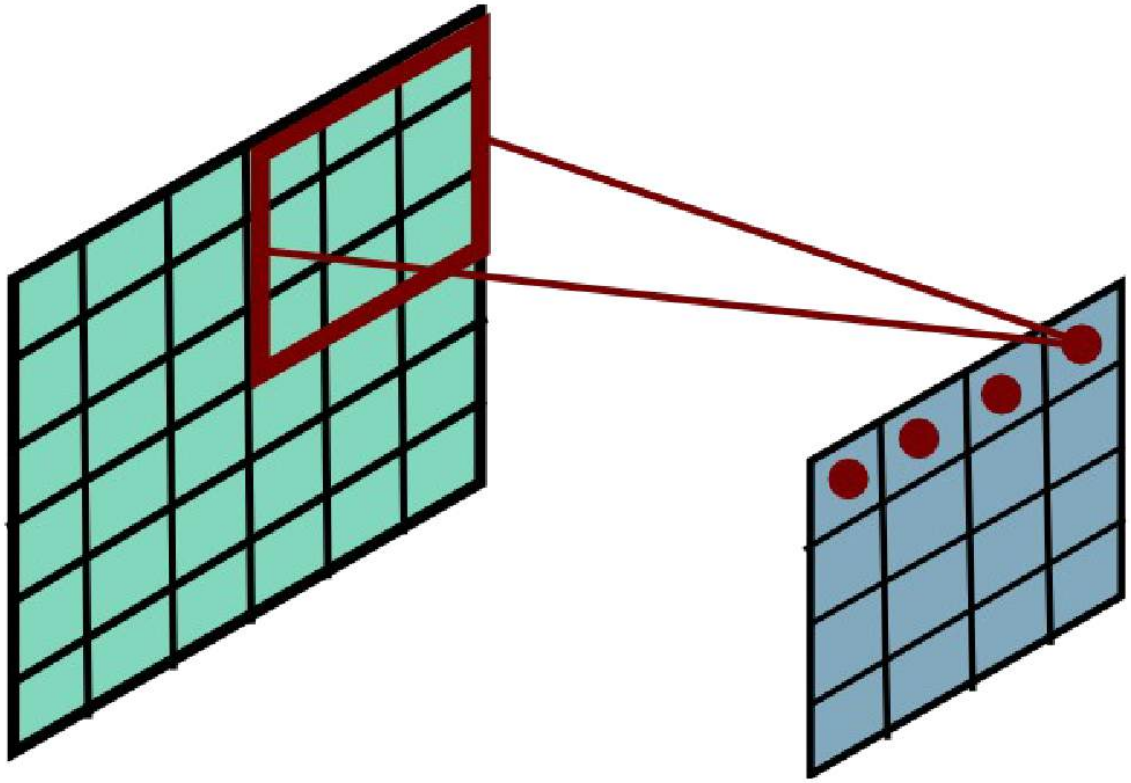
Convolutional Layer



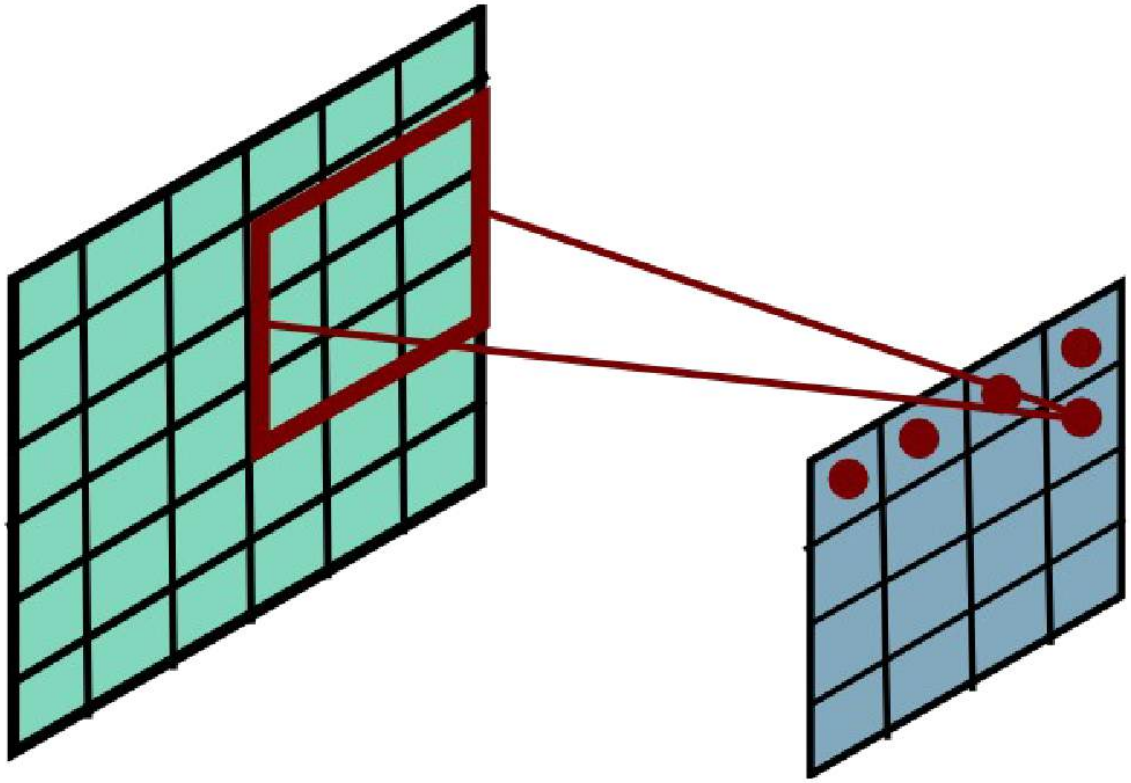
Convolutional Layer



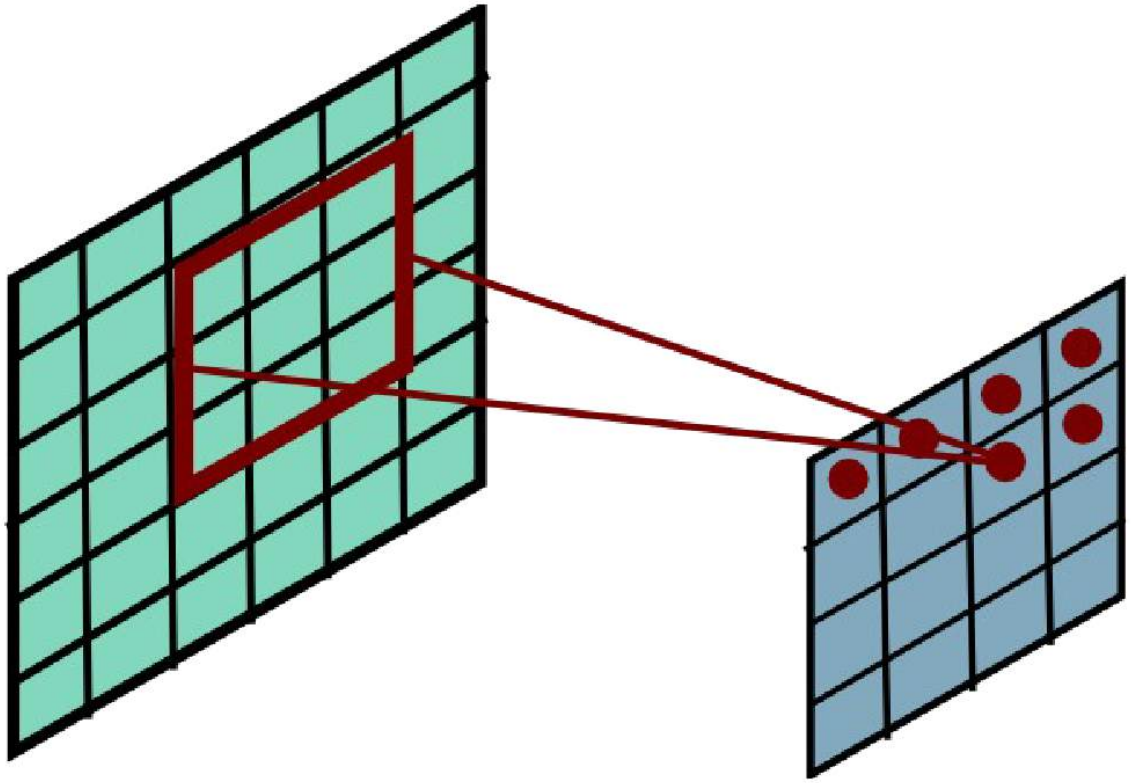
Convolutional Layer



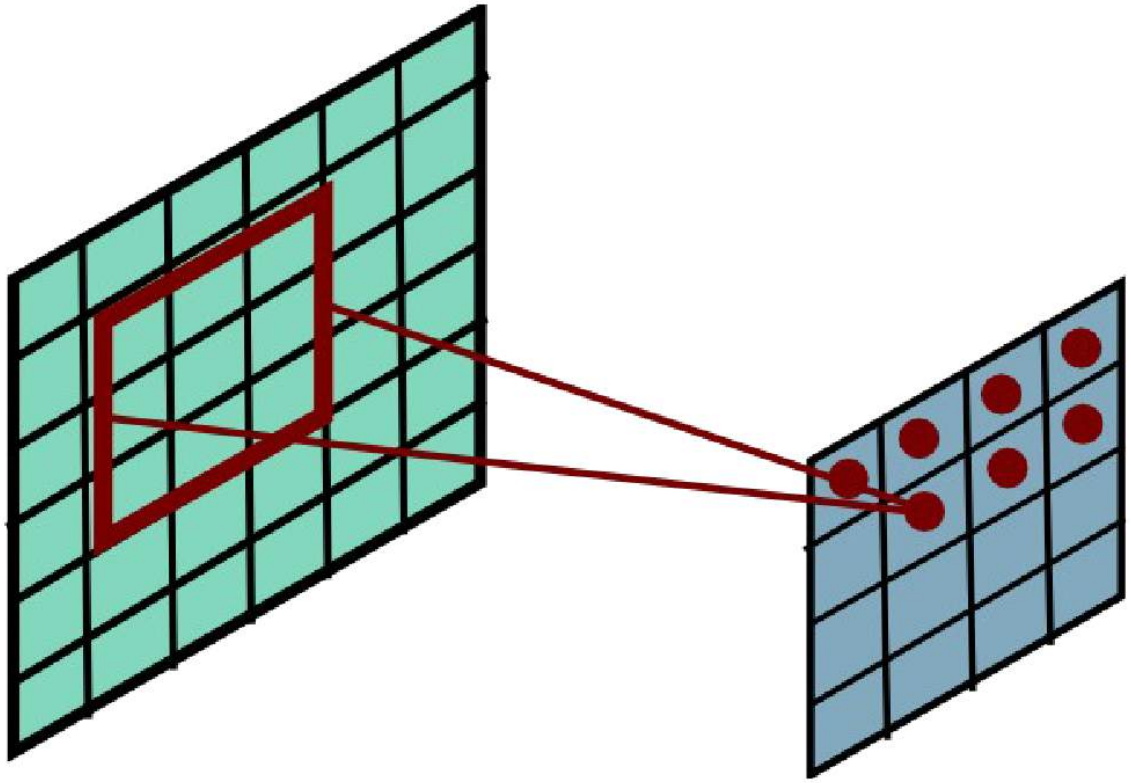
Convolutional Layer



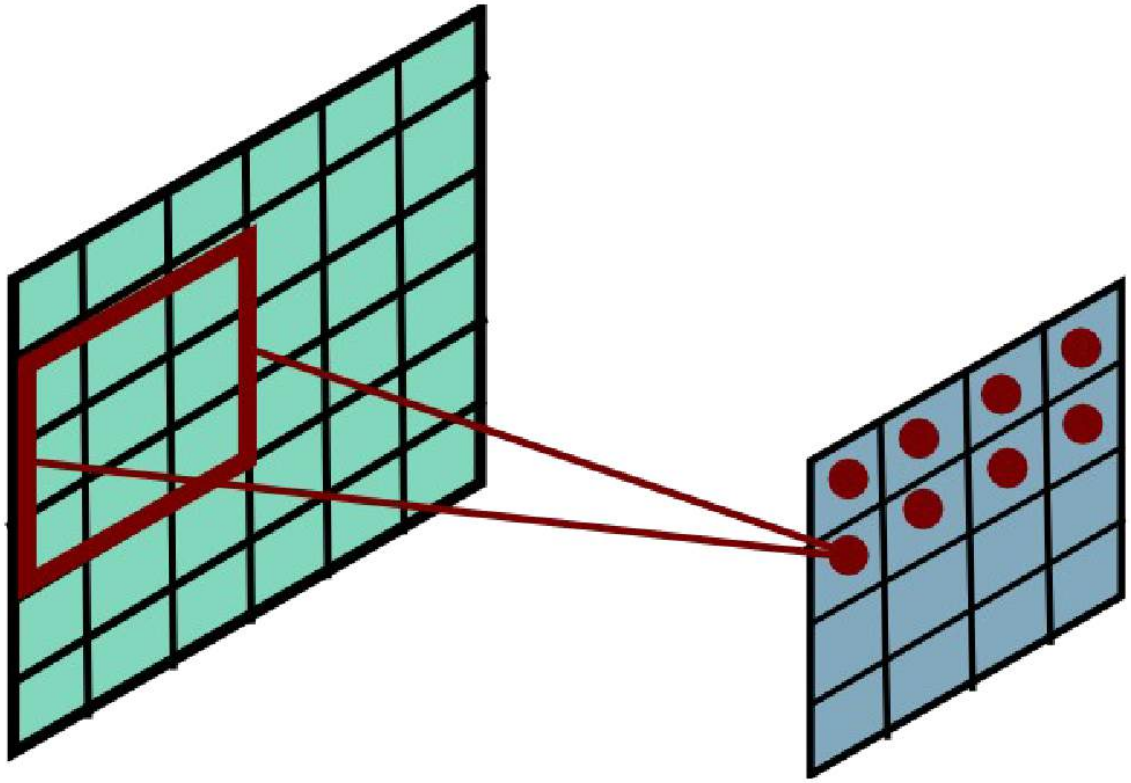
Convolutional Layer



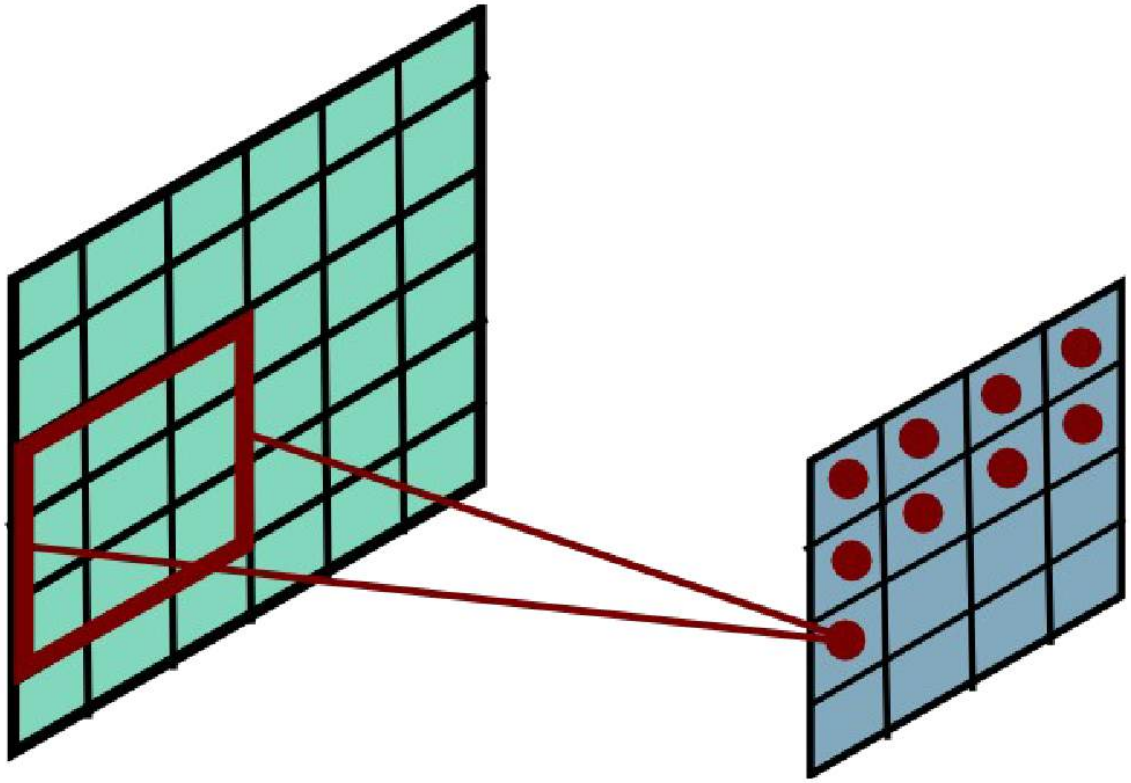
Convolutional Layer



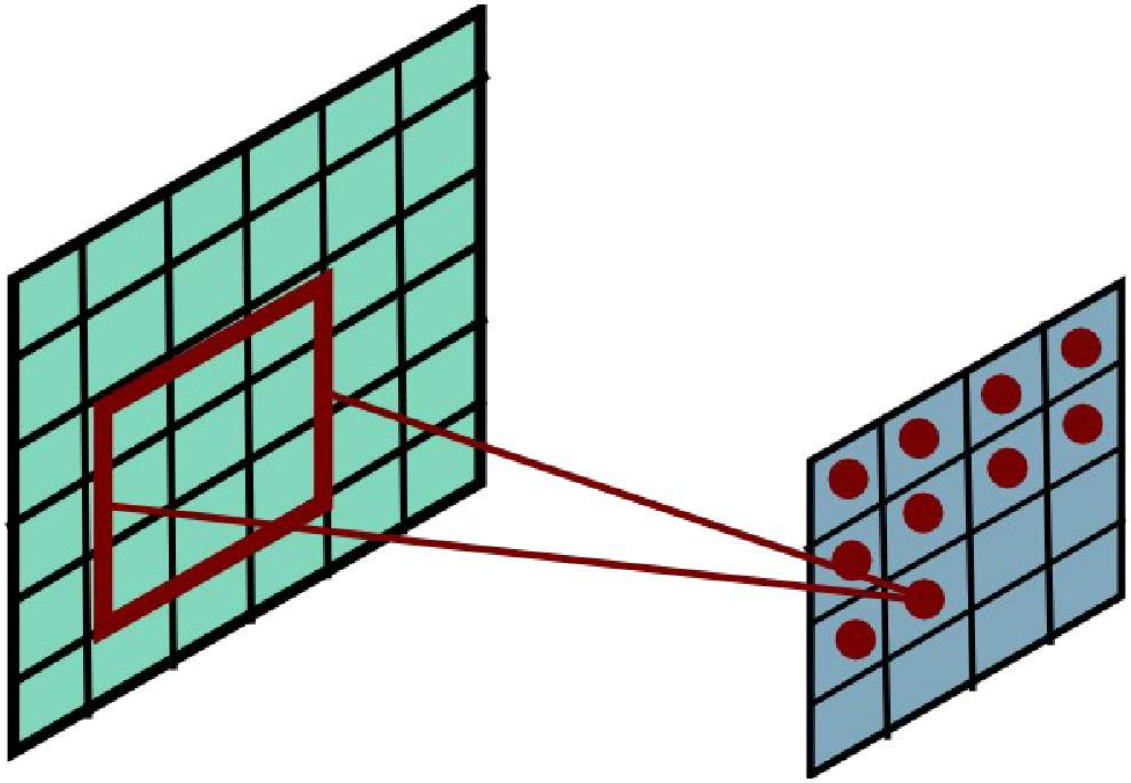
Convolutional Layer



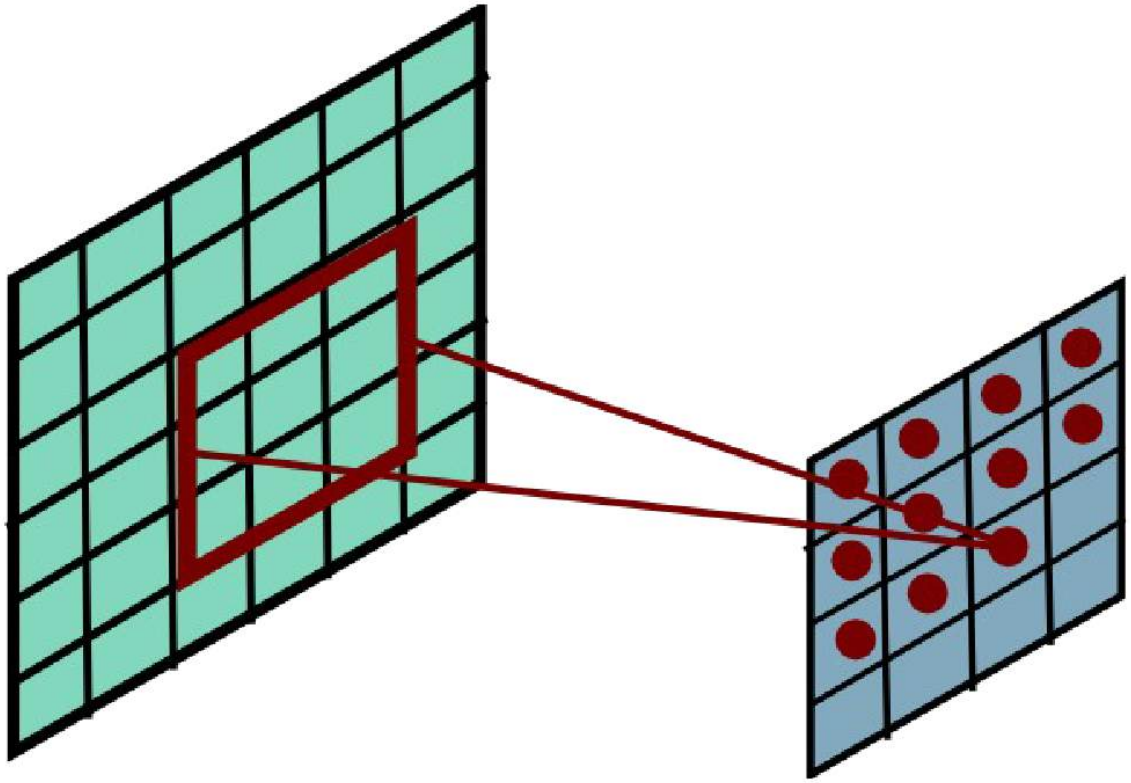
Convolutional Layer



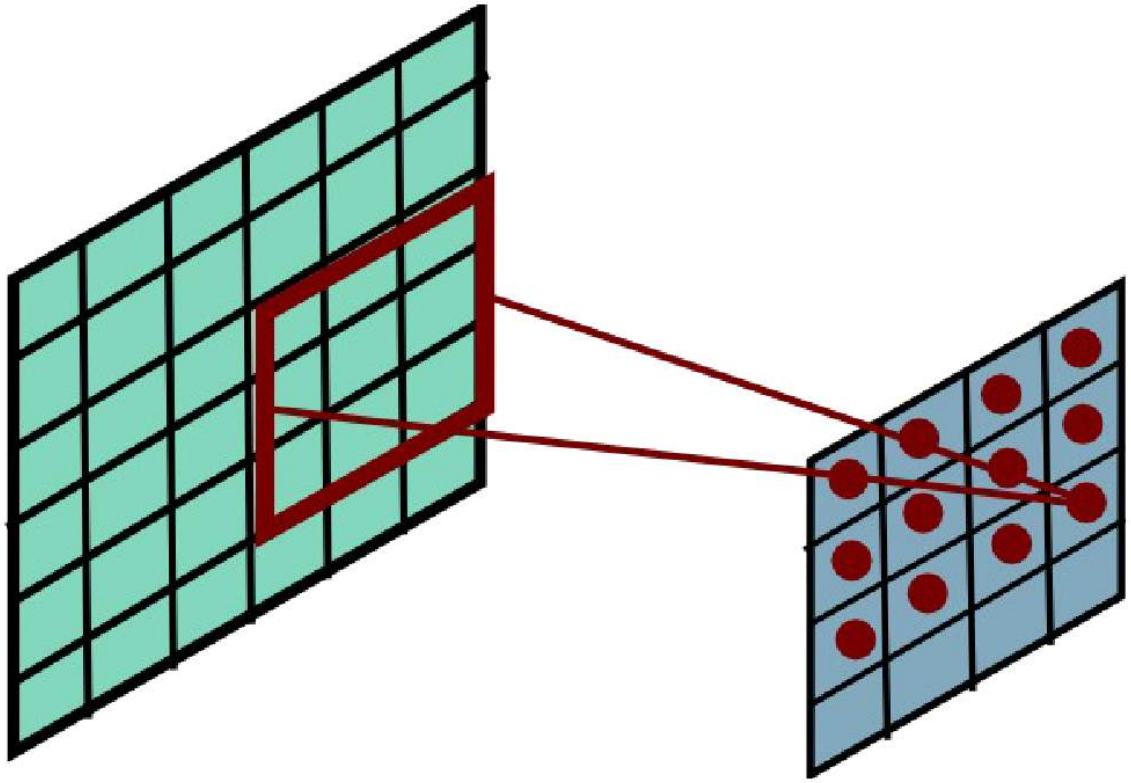
Convolutional Layer



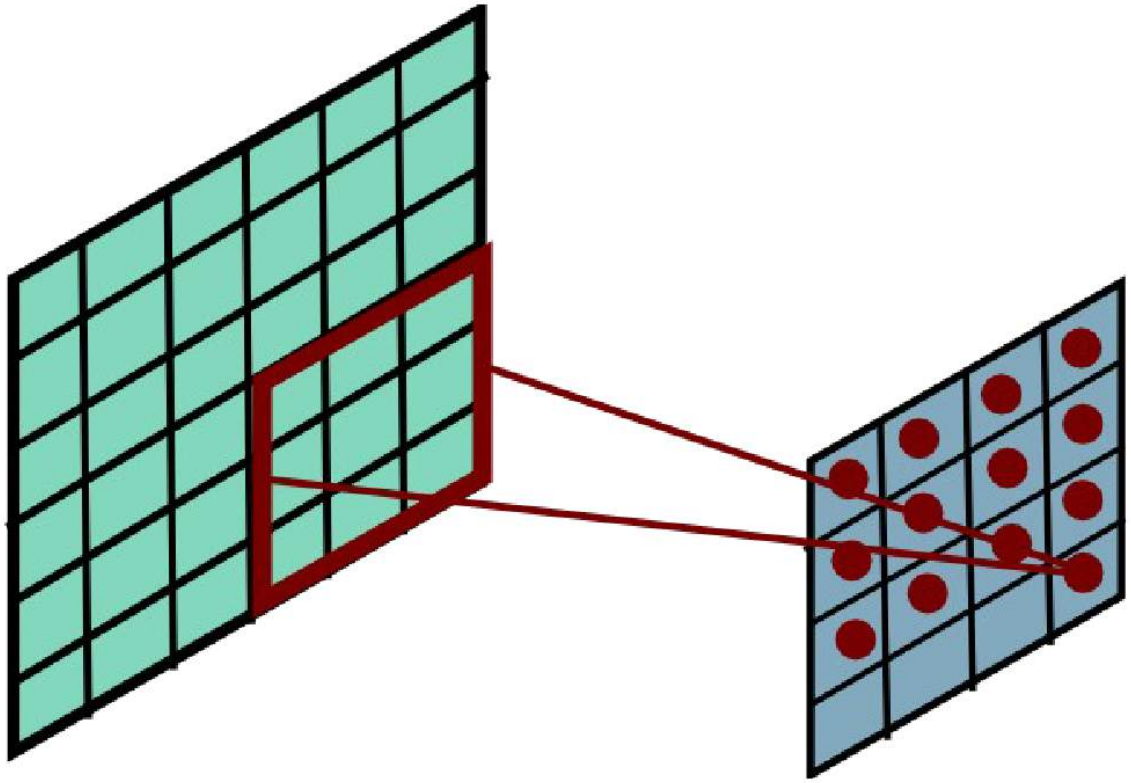
Convolutional Layer



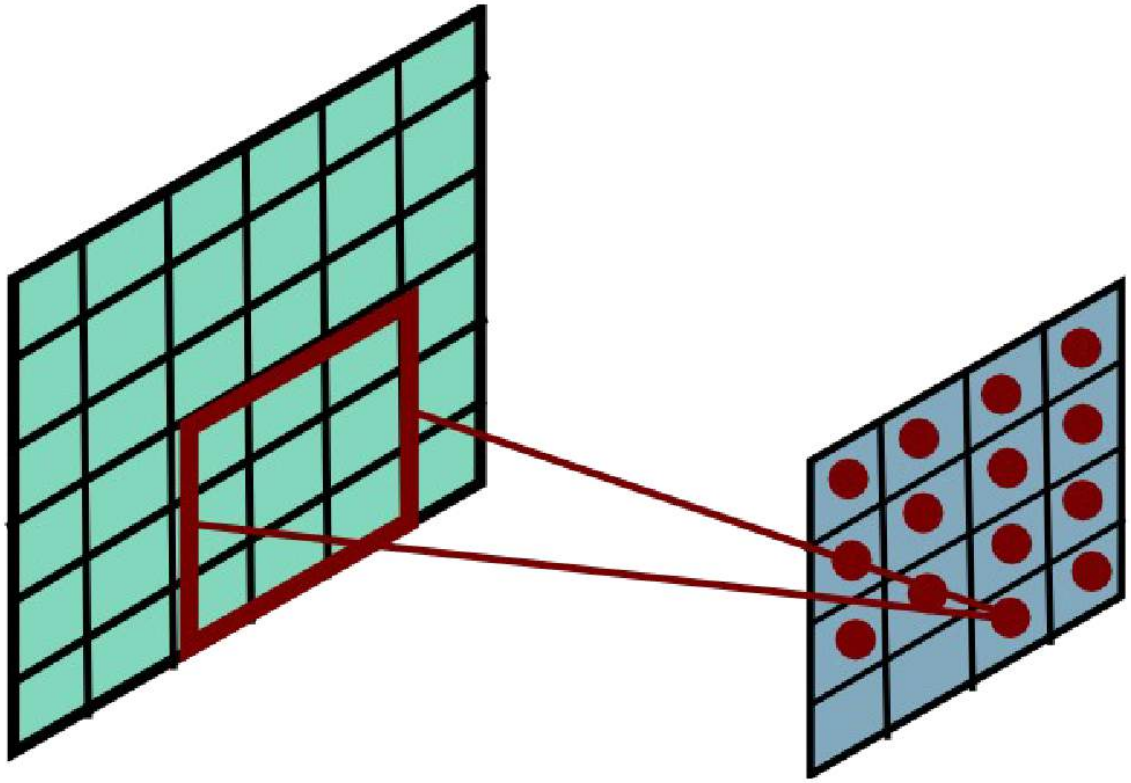
Convolutional Layer



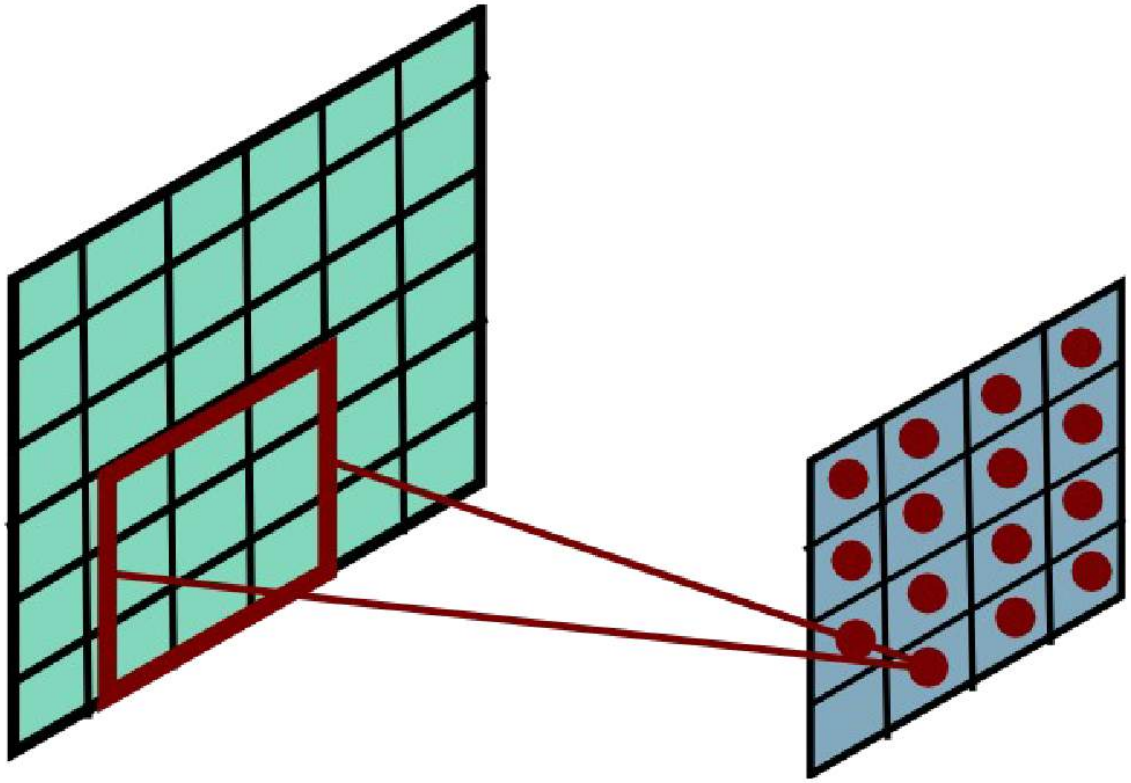
Convolutional Layer



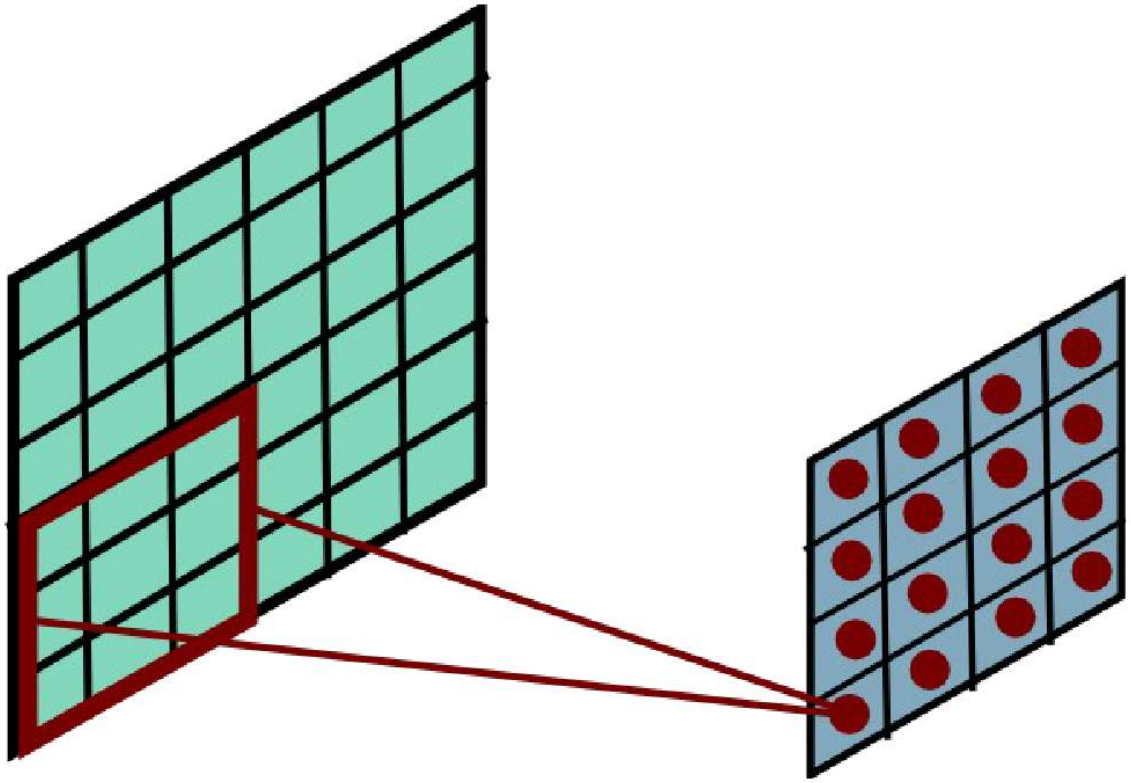
Convolutional Layer



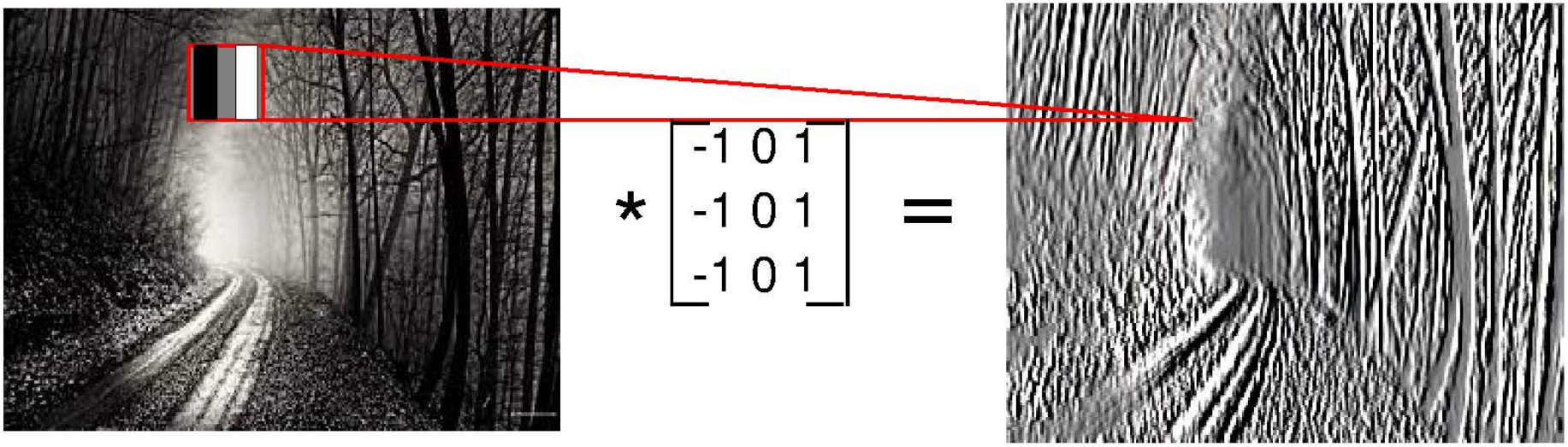
Convolutional Layer



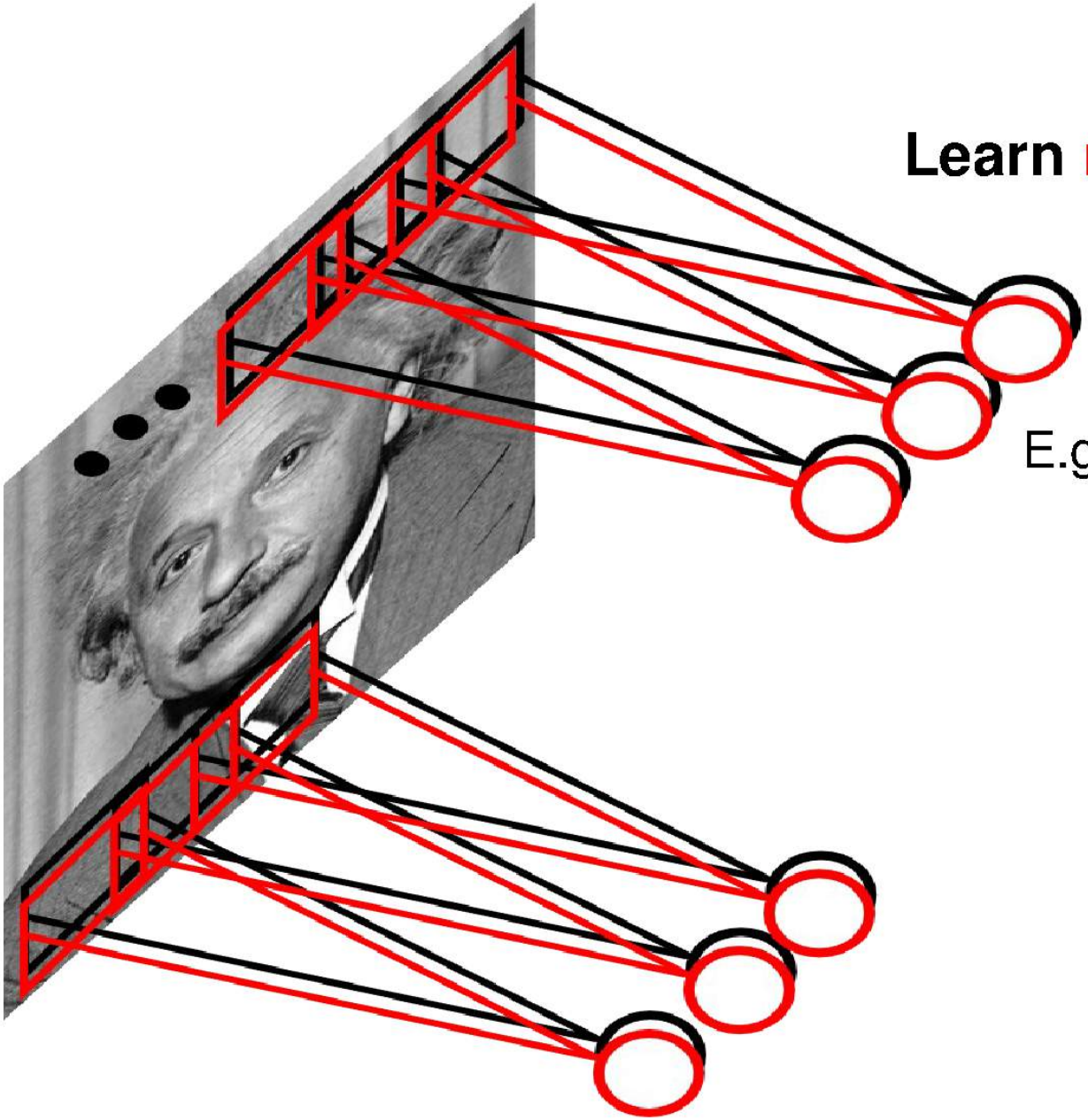
Convolutional Layer



Convolutional Layer



Convolutional Layer



Learn multiple filters.

E.g.: 200x200 image
100 Filters
Filter size: 10x10
10K parameters

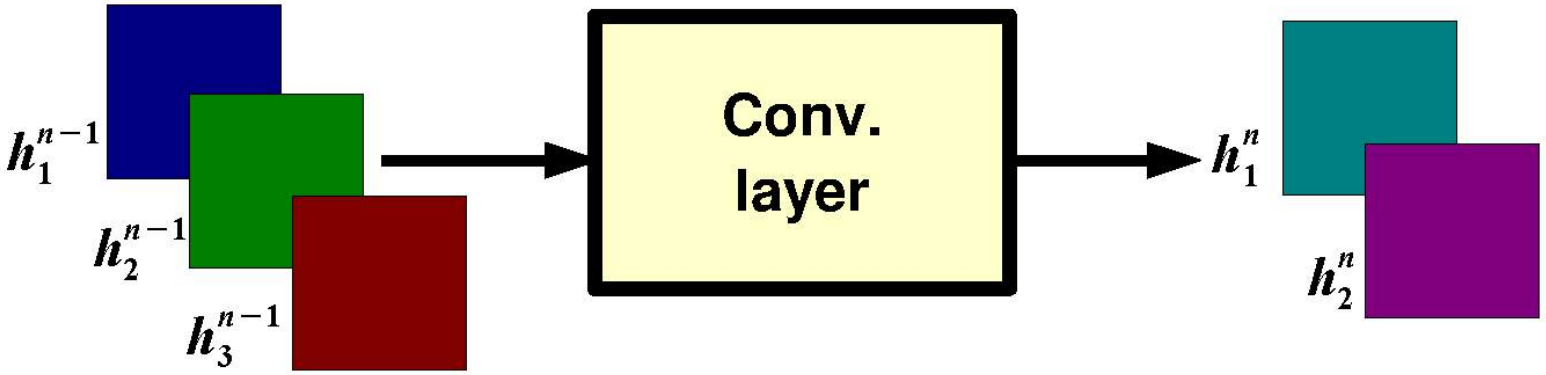
Convolutional Layer

$$h_j^n = max\left(0, \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

output feature map

input feature map

kernel



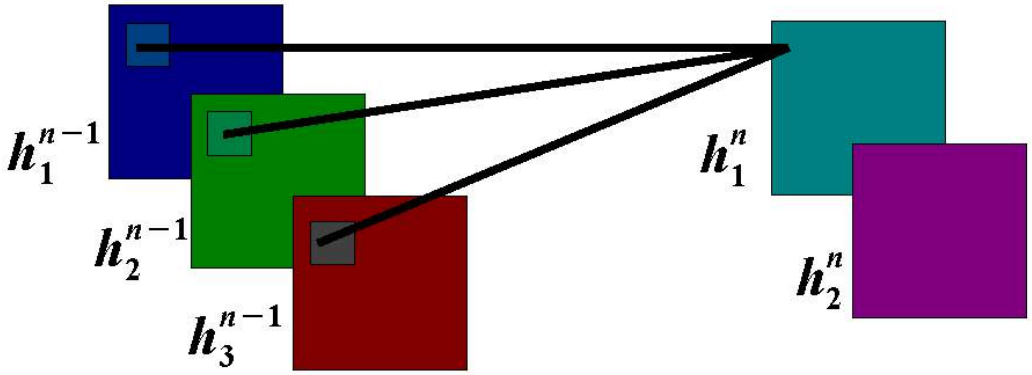
Convolutional Layer

$$h_j^n = max\left(0, \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

output feature map

input feature map

kernel



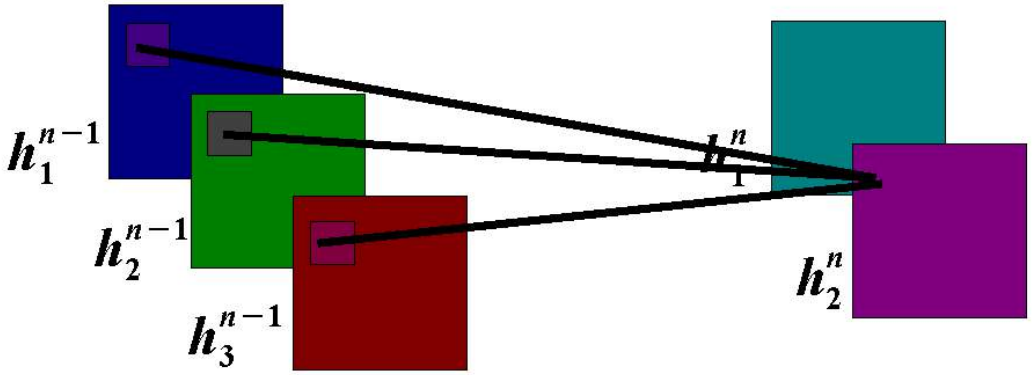
Convolutional Layer

$$h_j^n = max\left(0, \sum_{k=1}^{K} h_k^{n-1} * w_{kj}^n\right)$$

output feature map

input feature map

kernel



Convolutional Layer

Question: Size of the output? Computational cost?

Answer: Depends on the number of filters and the stride. Example: Kernels size KxK, input size DxD, stride 1, M input feature maps & N output feature maps

- input size MxDxD, output size Nx(D-K+1)x(D-K+1)
- kernels size MxNxKxK (coefficients have to be learned)
- cost: M*K*K*N*(D-K+1)*(D-K+1)

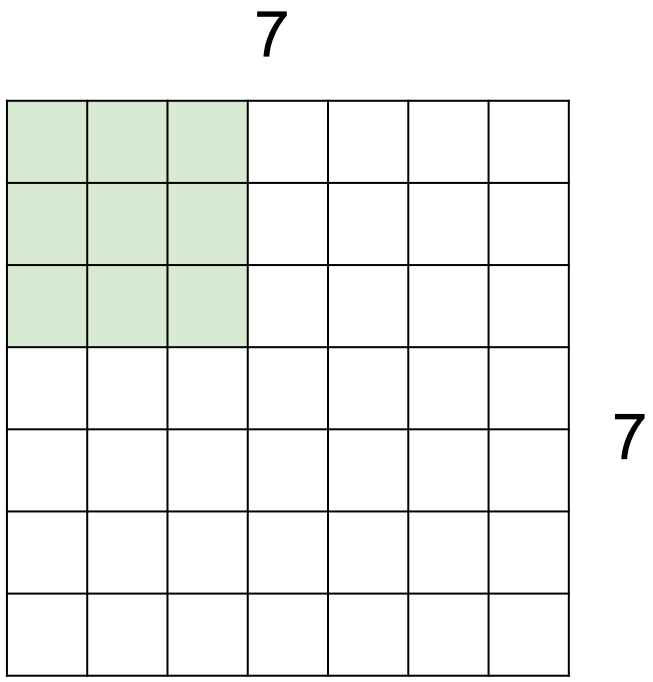
Question: How many feature maps? What's the size of the filters?

Answer: Usually, output feature maps > input feature maps. The size of the filters has to match the size/scale of the patterns we want to detect (task dependent).

Question: How to compute the gradients?

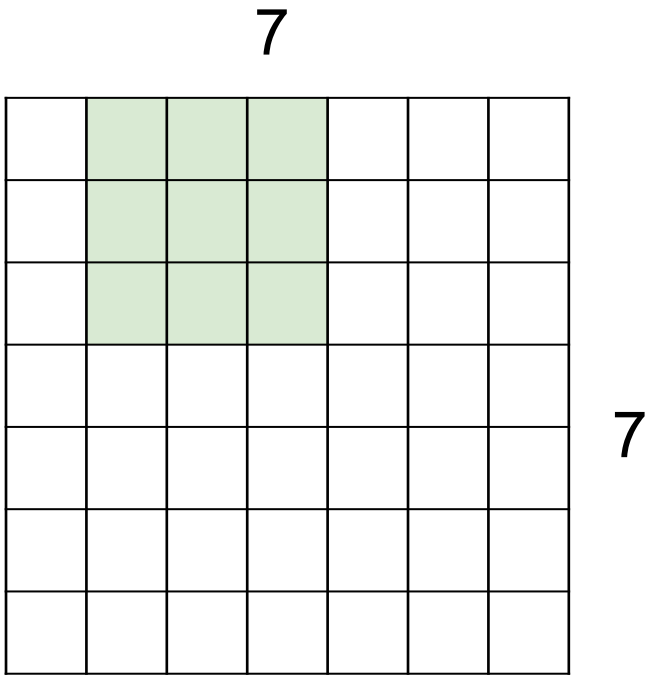
Hint: Convolutions are linear operations

Convolutional Layer: Spatial Dimensions



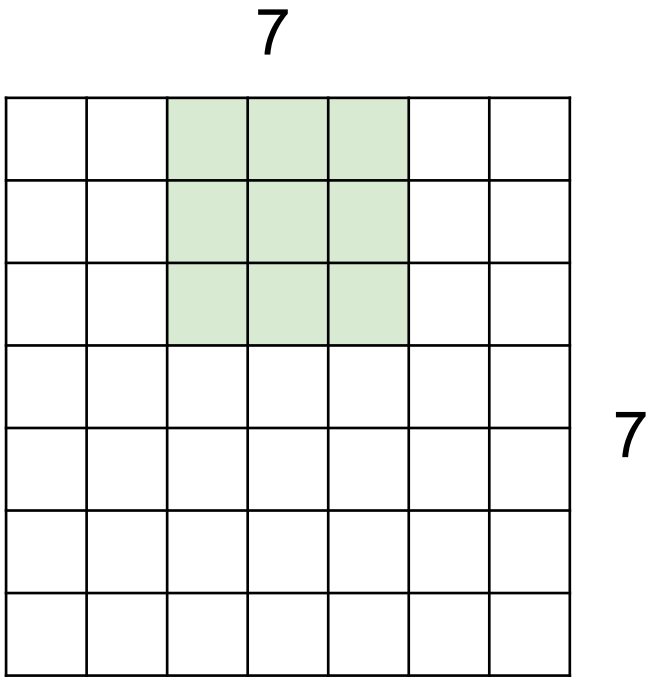
7x7 input (spatially)
assume 3x3 filter

Convolutional Layer: Spatial Dimensions



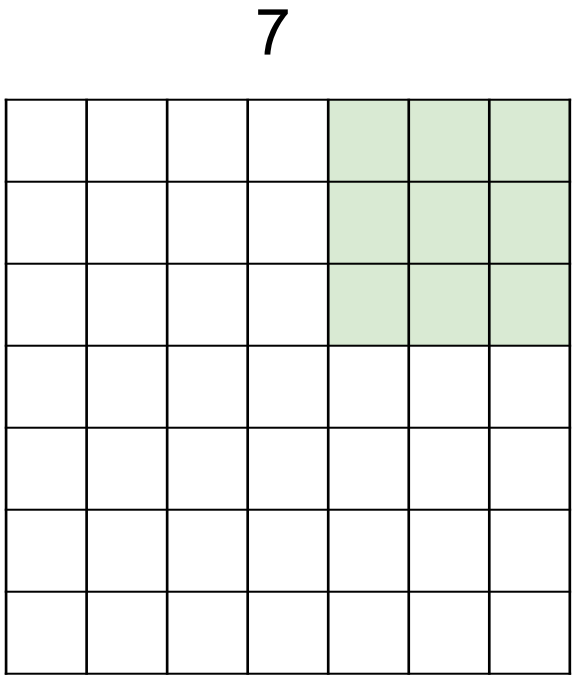
7x7 input (spatially)
assume 3x3 filter

Convolutional Layer: Spatial Dimensions



7x7 input (spatially)
assume 3x3 filter

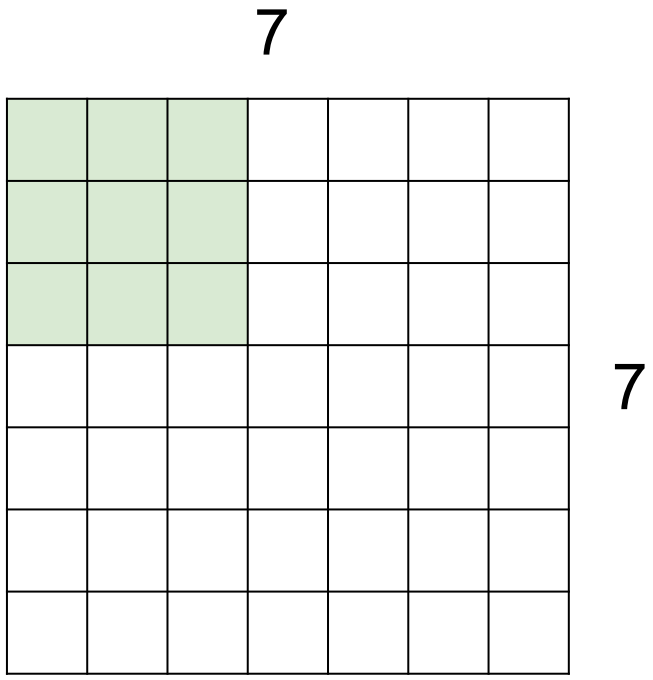
Convolutional Layer: Spatial Dimensions



7x7 input (spatially)
assume 3x3 filter

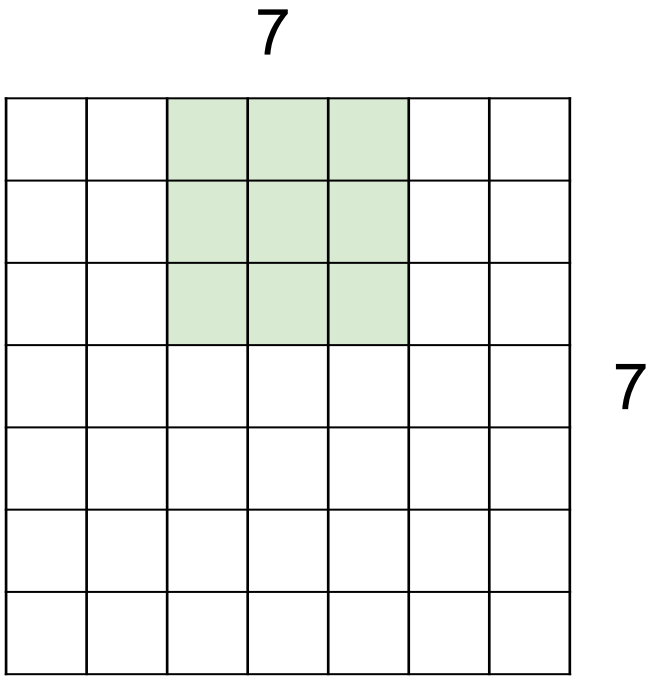
=> 5x5 output

Convolutional Layer: Spatial Dimensions



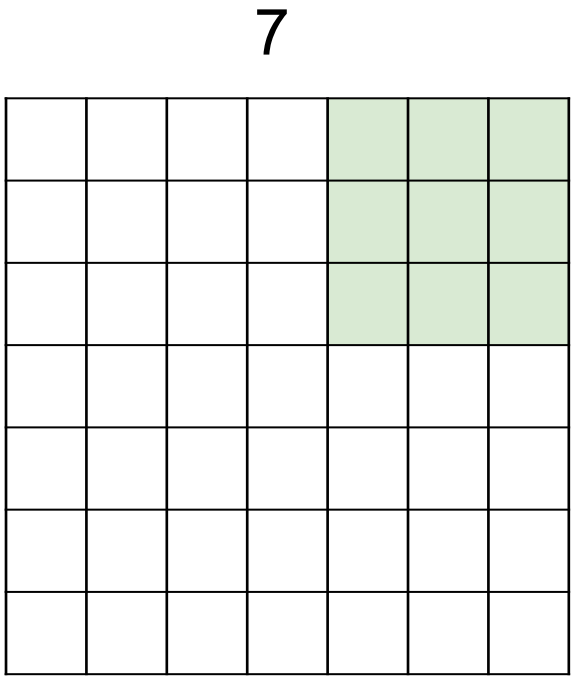
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Convolutional Layer: Spatial Dimensions



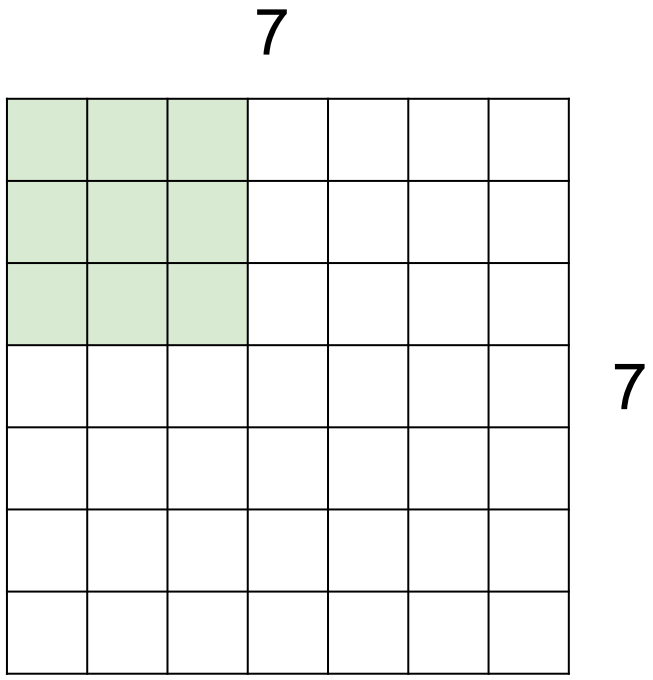
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

Convolutional Layer: Spatial Dimensions



7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output

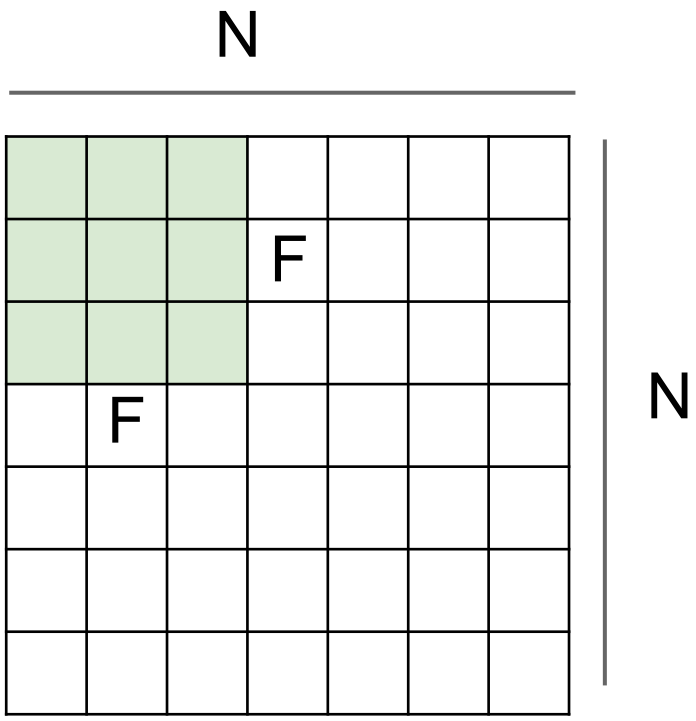
Convolutional Layer: Spatial Dimensions



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3**

doesn't fit!
cannot apply 3x3 filter on 7x7 input with stride 3.

Convolutional Layer: Spatial Dimensions

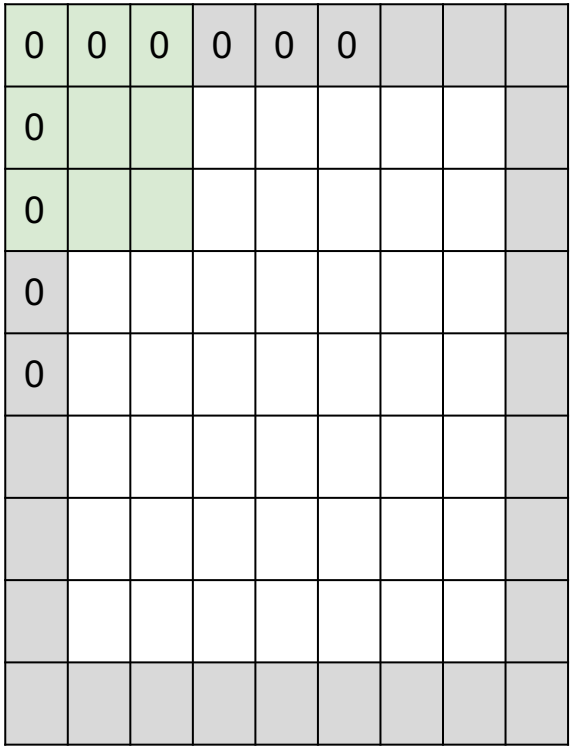


Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33 ☹

Convolutional Layer: Spatial Dimensions

In practice: Common to zero pad the border



e.g. input 7x7

3x3 filter, applied with **stride 1**

pad with 1 pixel border => what is the output?

(recall:)

(N - F) / stride + 1

Convolutional Layer: Recap

A standard neural net applied to images:

- scales quadratically with the size of the input
- does not leverage stationarity

Solution:

- connect each hidden unit to a small patch of the input
- share the weight across space
- still a linear function!

This is called: **convolutional layer / operation**

A network with convolutional layers: **convolutional network**

LeCun et al. “Gradient-based learning applied to document recognition” IEEE 1998

Convolutional Layer: Backprop

Question: How to compute the gradients in theory?

Hint: Convolutions are linear operations

Let us consider a 1D example $y = h * x$

$$y = h * x = \begin{bmatrix} h_1 & 0 & \dots & 0 & 0 \\ h_2 & h_1 & \dots & \vdots & \vdots \\ h_3 & h_2 & \dots & 0 & 0 \\ \vdots & h_3 & \dots & h_1 & 0 \\ h_{m-1} & \vdots & \dots & h_2 & h_1 \\ h_m & h_{m-1} & \vdots & \vdots & h_2 \\ 0 & h_m & \dots & h_{m-2} & \vdots \\ 0 & 0 & \dots & h_{m-1} & h_{m-2} \\ \vdots & \vdots & \vdots & h_m & h_{m-1} \\ 0 & 0 & 0 & \dots & h_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

H Toeplitz matrix

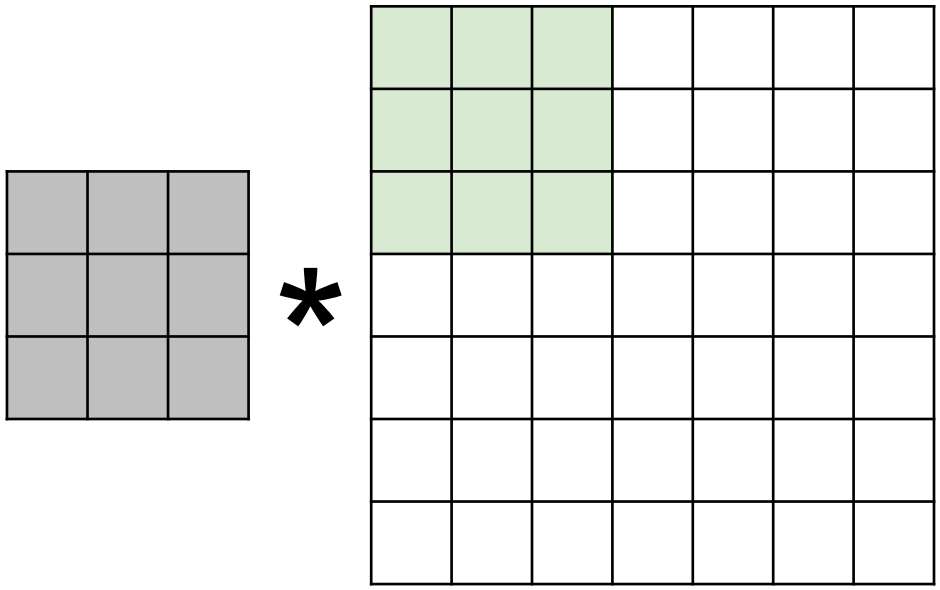
$$y = Hx$$

$$\frac{dy}{dx} = H, \frac{dy}{dH} = x^T \otimes I$$

Convolutional Layer: Backprop

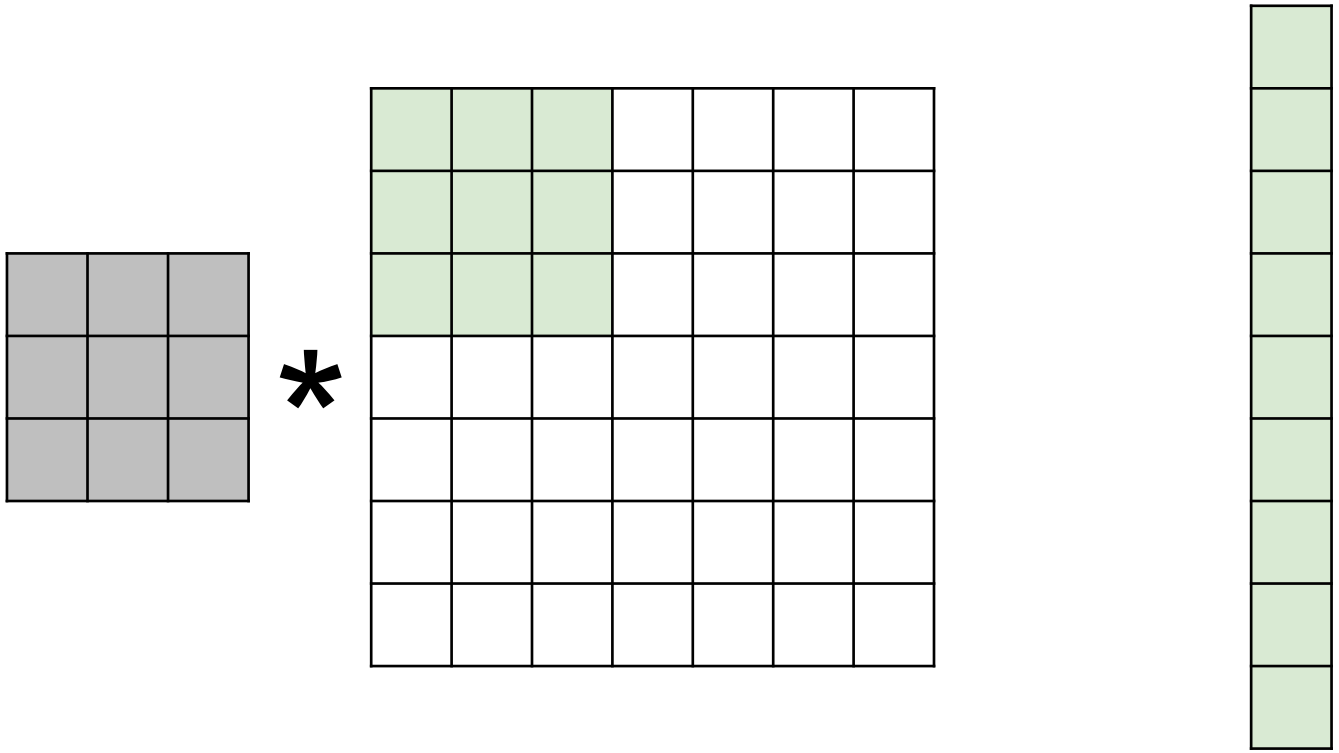
Question: How to compute the gradients in practice?

Hint: Unfold the image & use efficient matrix multiplication



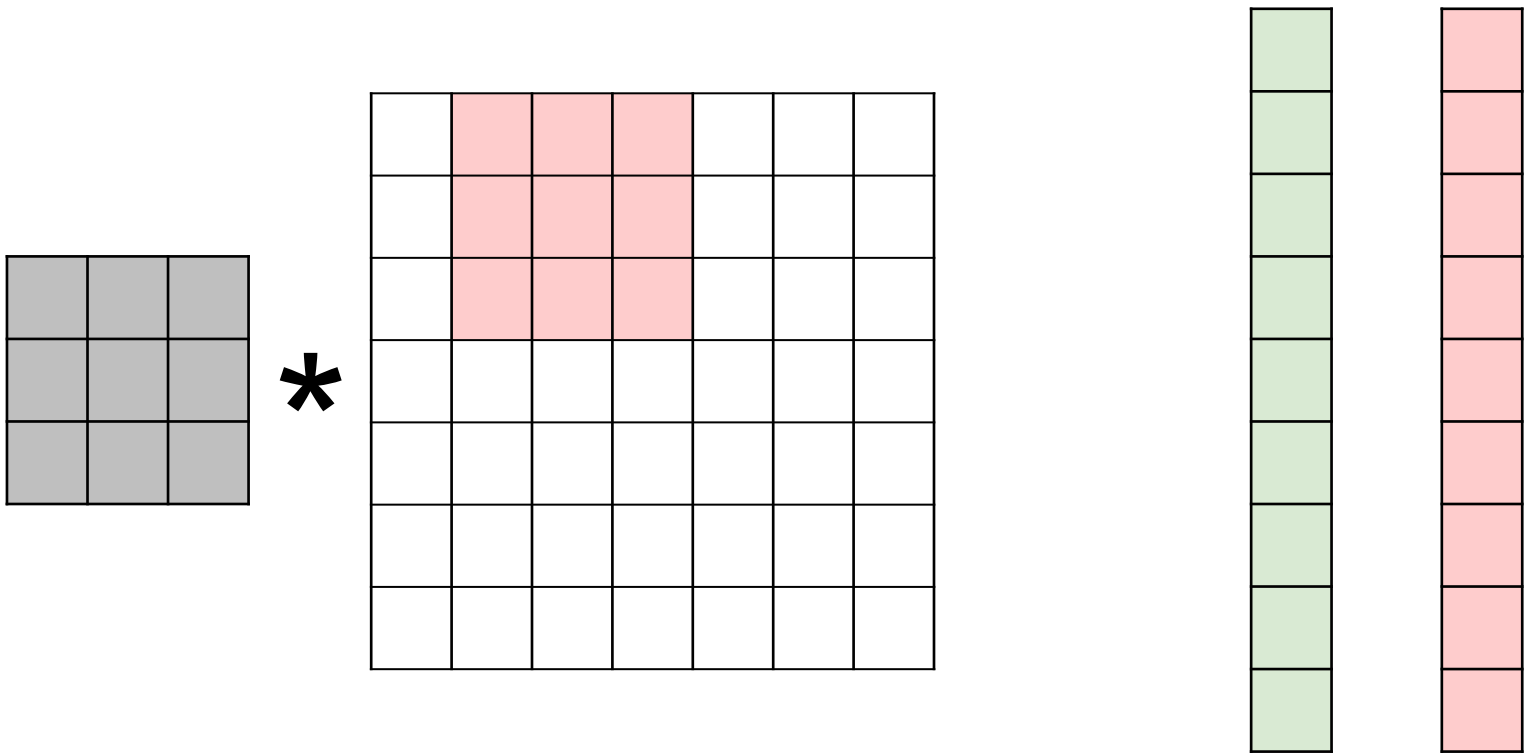
Convolutional Layer: Backprop

Unfold the image & use efficient matrix multiplication



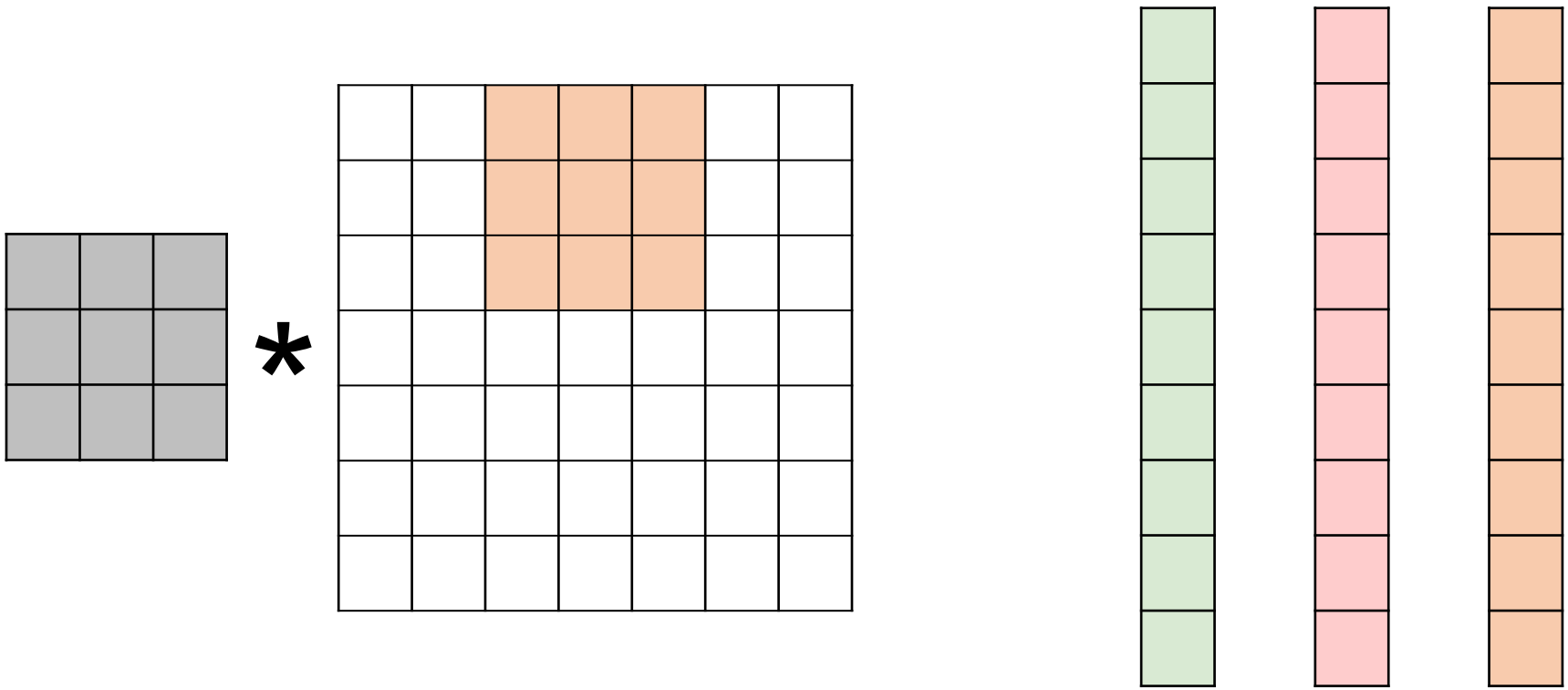
Convolutional Layer: Backprop

Unfold the image & use efficient matrix multiplication



Convolutional Layer: Backprop

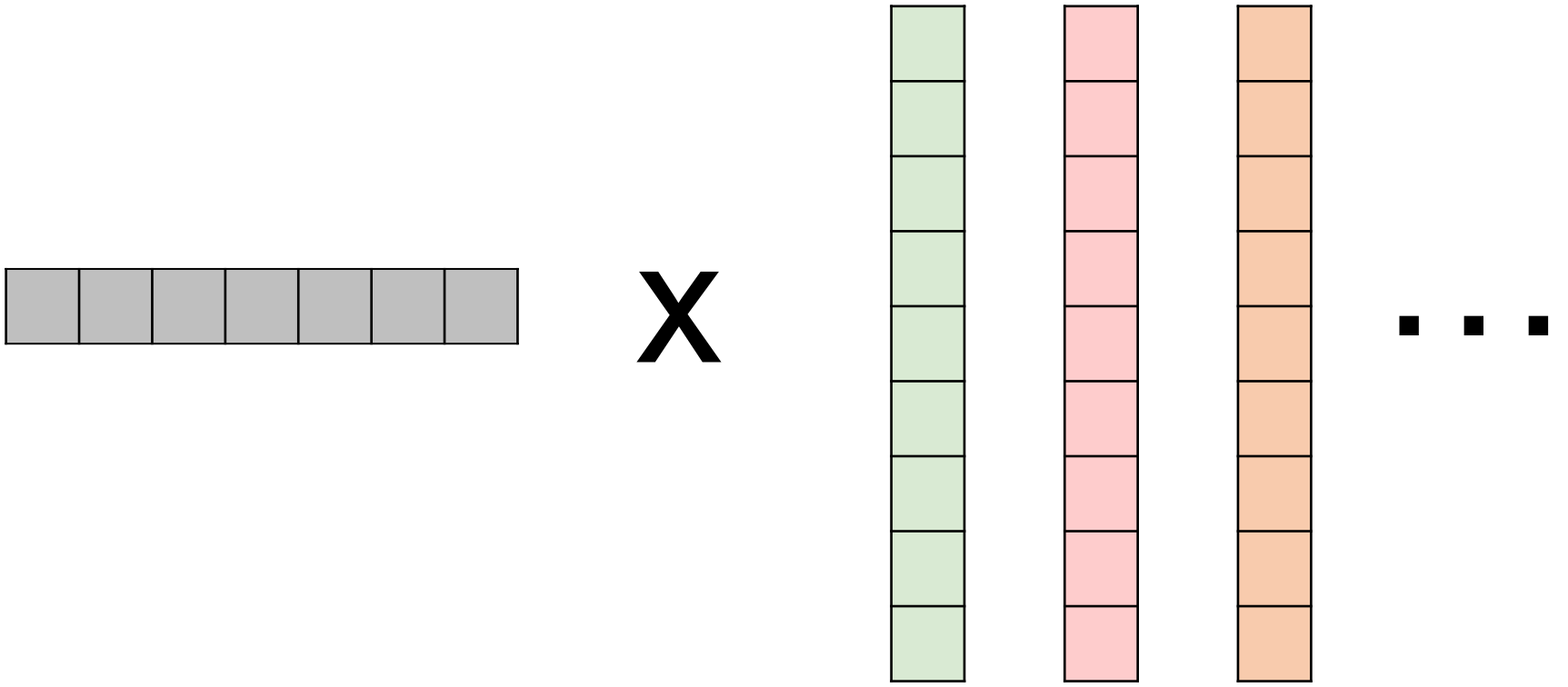
Unfold the image & use efficient matrix multiplication



Convolutional Layer: Backprop

Unfold the image & use efficient matrix multiplication

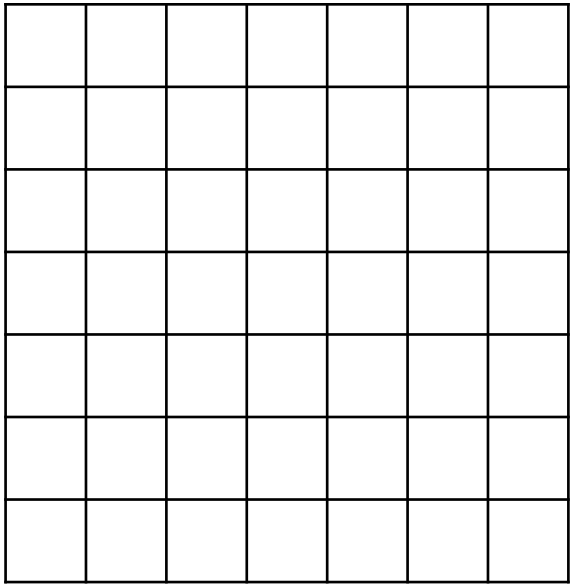
Unfold: Can have arbitrary strides

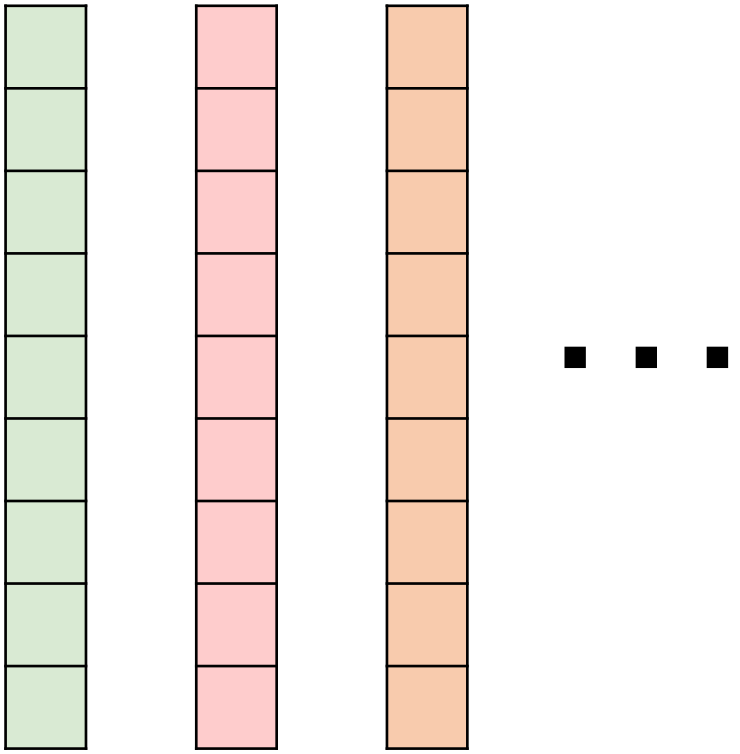


Convolutional Layer: Backprop

Fold the vectors back into an image

Fold: Can have arbitrary strides

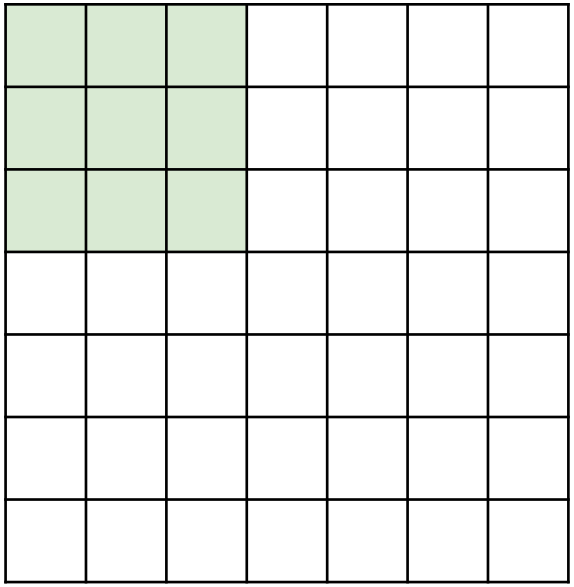


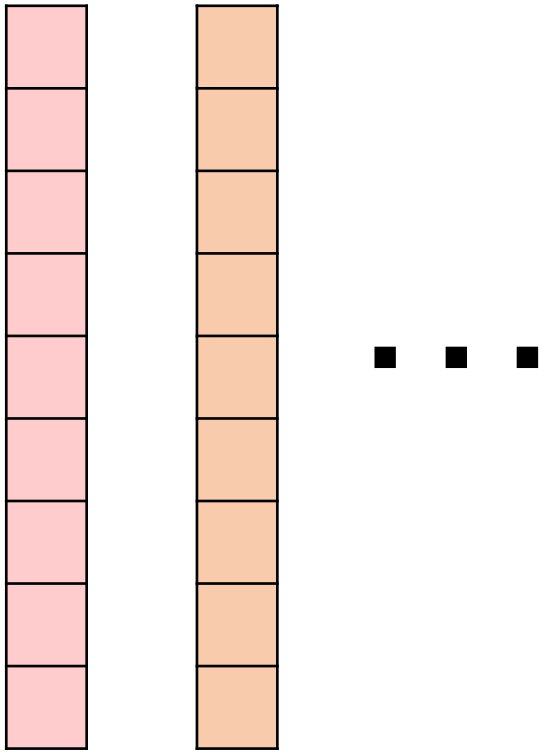


Convolutional Layer: Backprop

Fold the vectors back into an image

Fold: Can have arbitrary strides

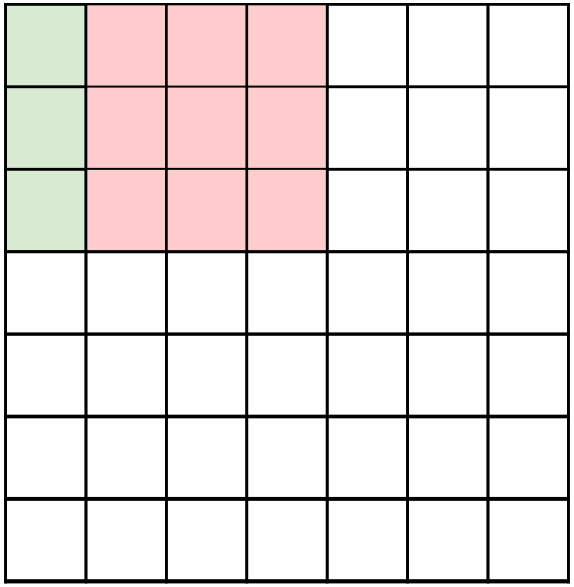


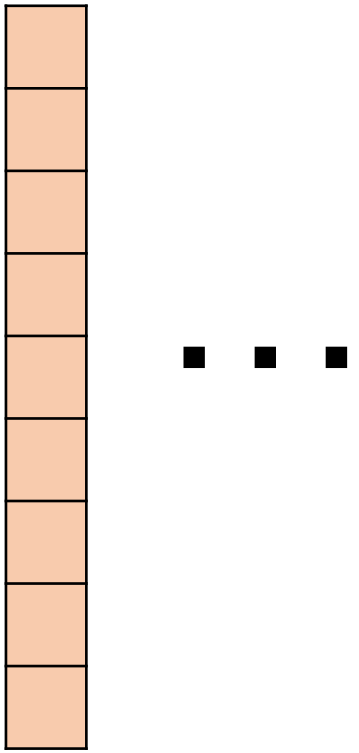


Convolutional Layer: Backprop

Fold the vectors back into an image

Fold: Can have arbitrary strides

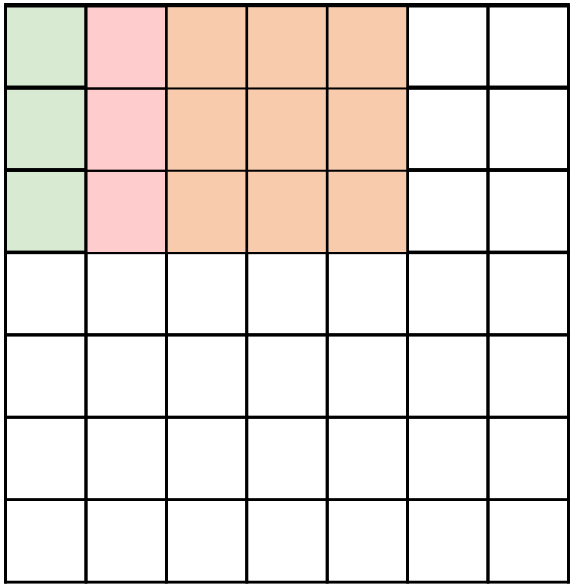




Convolutional Layer: Backprop

Fold the vectors back into an image

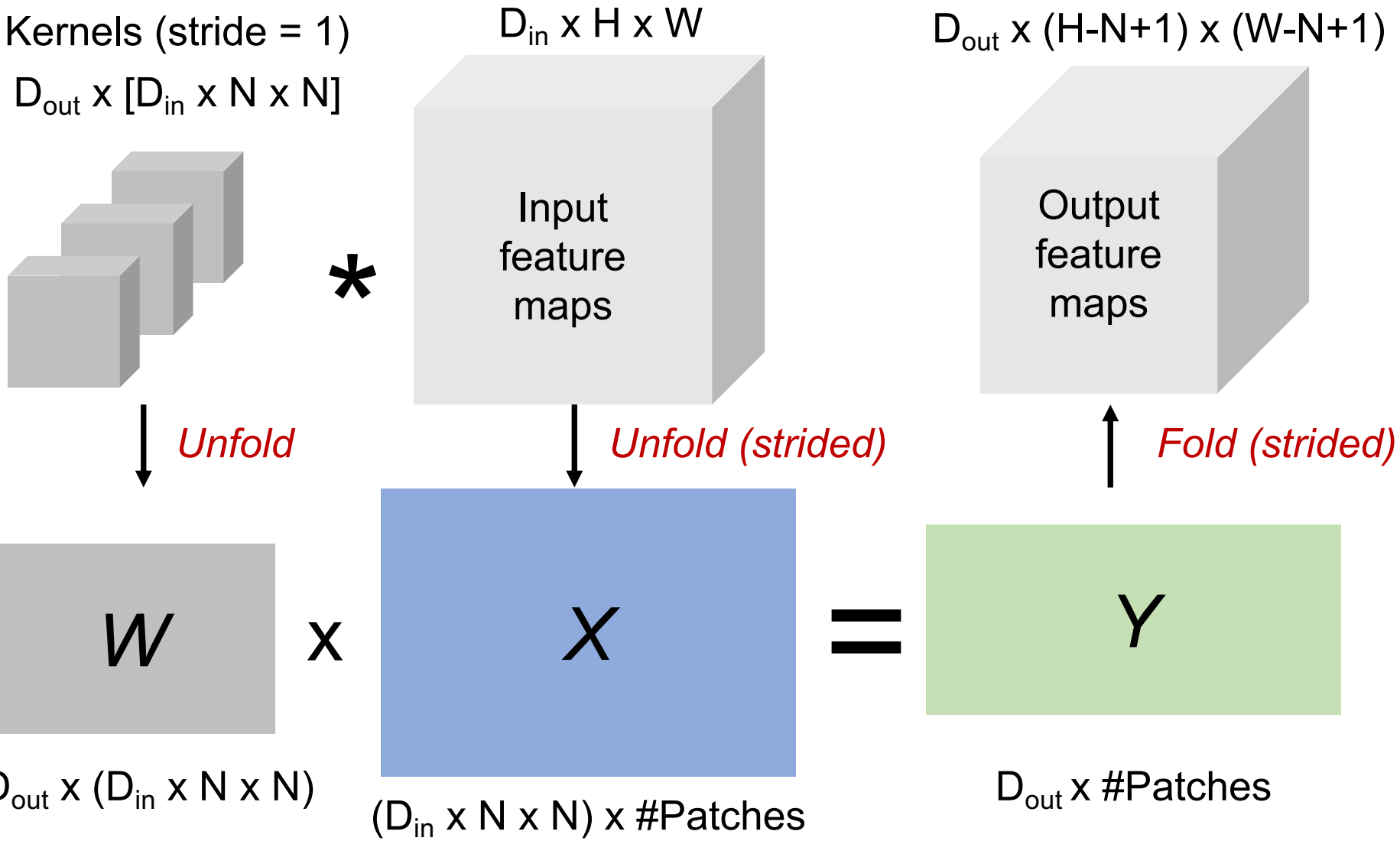
Fold: Can have arbitrary strides



. . .

Convolutional Layer: Backprop

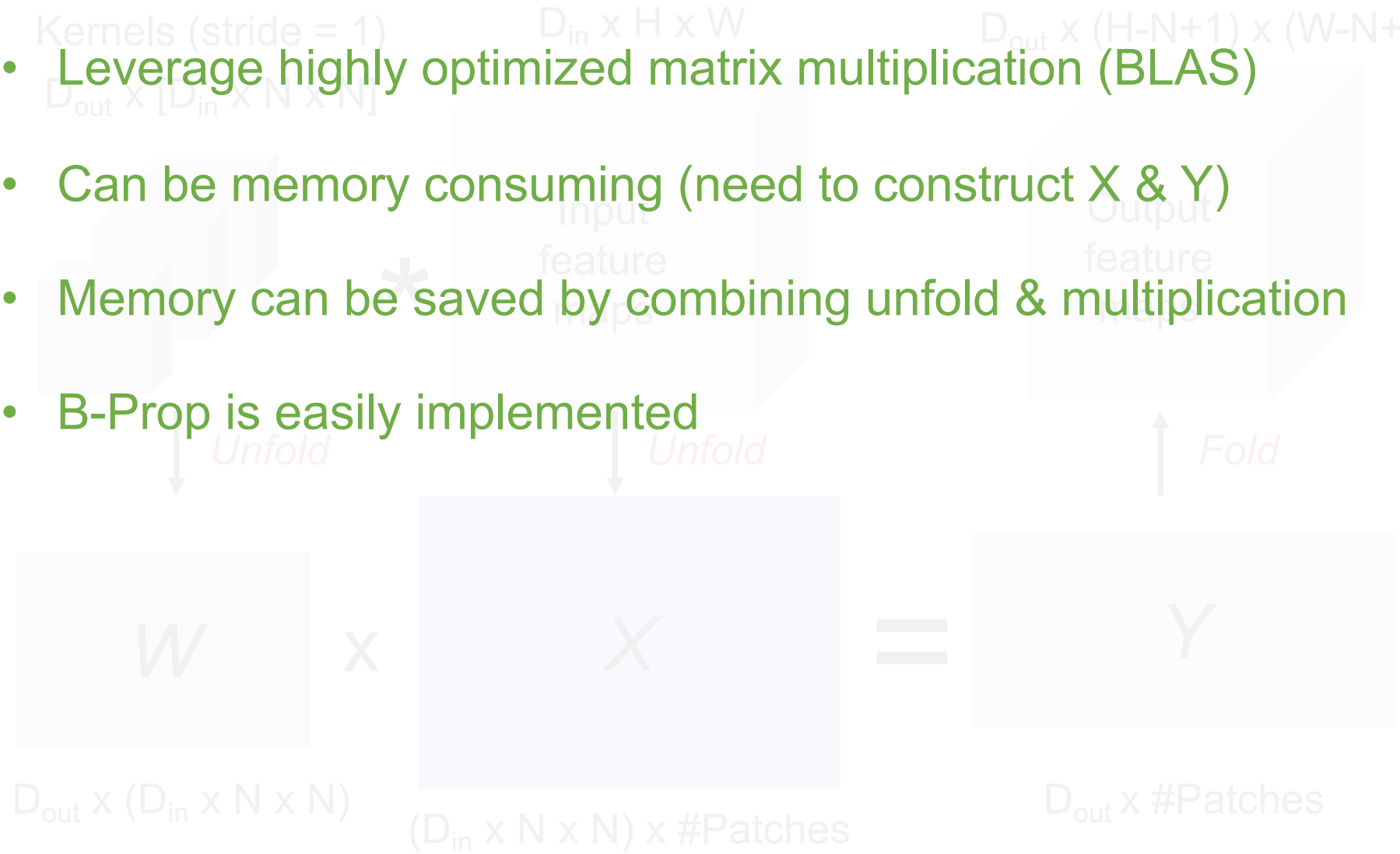
F-Prop: Unfold the image & use efficient matrix multiplication



Convolutional Layer: Backprop

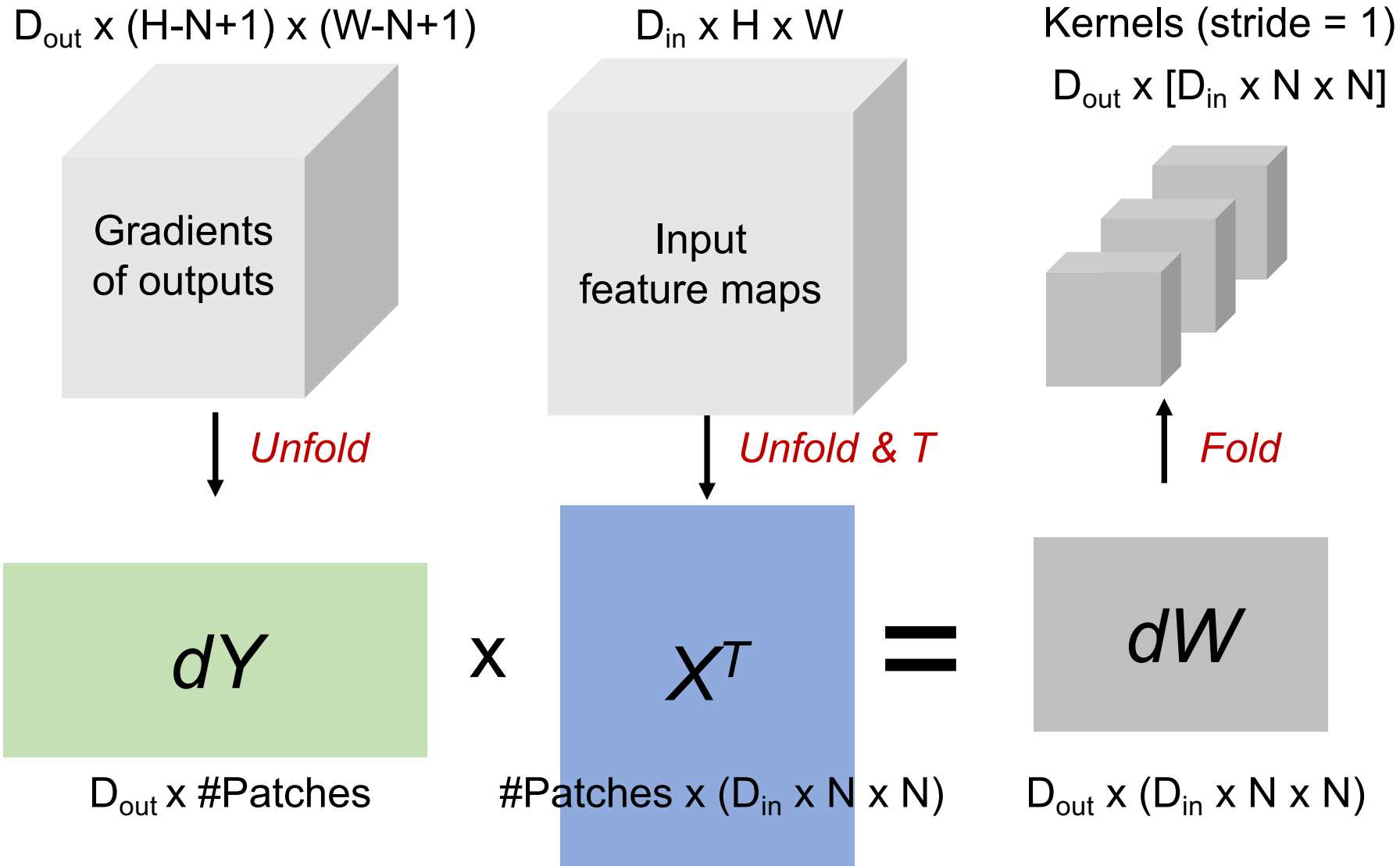
F-Prop: Unfold the image & use efficient matrix multiplication

- Leverage highly optimized matrix multiplication (BLAS)
- Can be memory consuming (need to construct X & Y)
- Memory can be saved by combining unfold & multiplication
- B-Prop is easily implemented



Convolutional Layer: Backprop

B-Prop: Compute gradients w.r.t. parameters

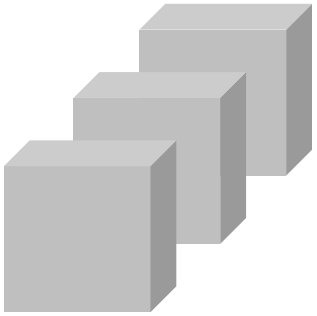


Convolutional Layer: Backprop

B-Prop: Compute gradients w.r.t. inputs

Kernels (stride = 1)

D_{out} x [D_{in} x N x N]



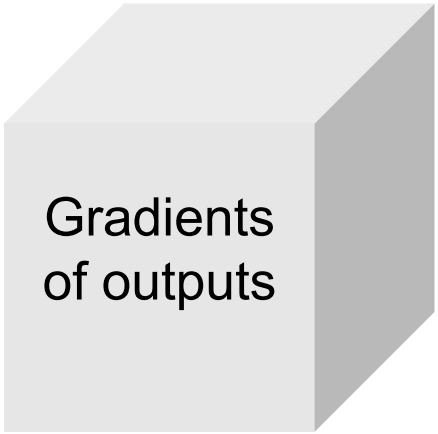
Unfold & T

W^T

(D_{in} x N x N) x D_{out}

x

D_{out} x (H-N+1) x (W-N+1)



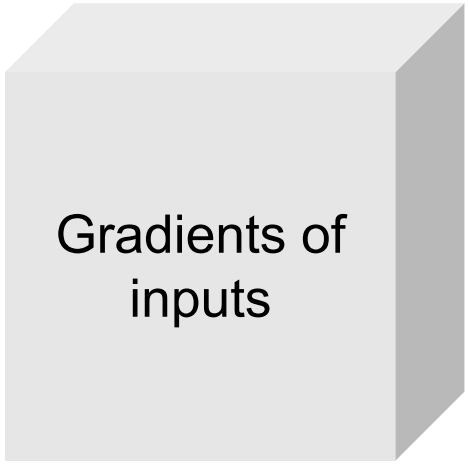
Unfold

dY

D_{out} x #Patches

=

D_{in} x H x W

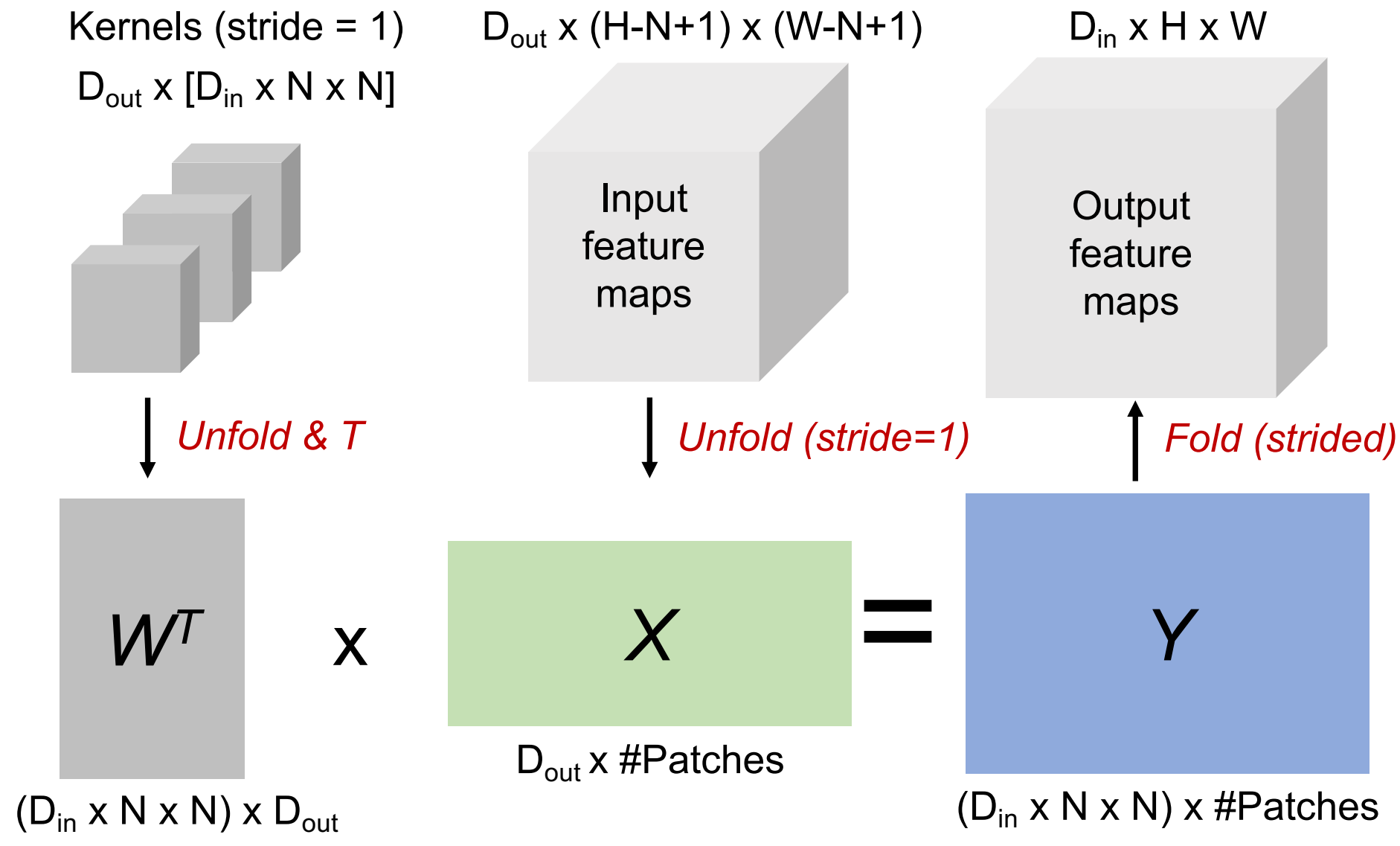


Fold

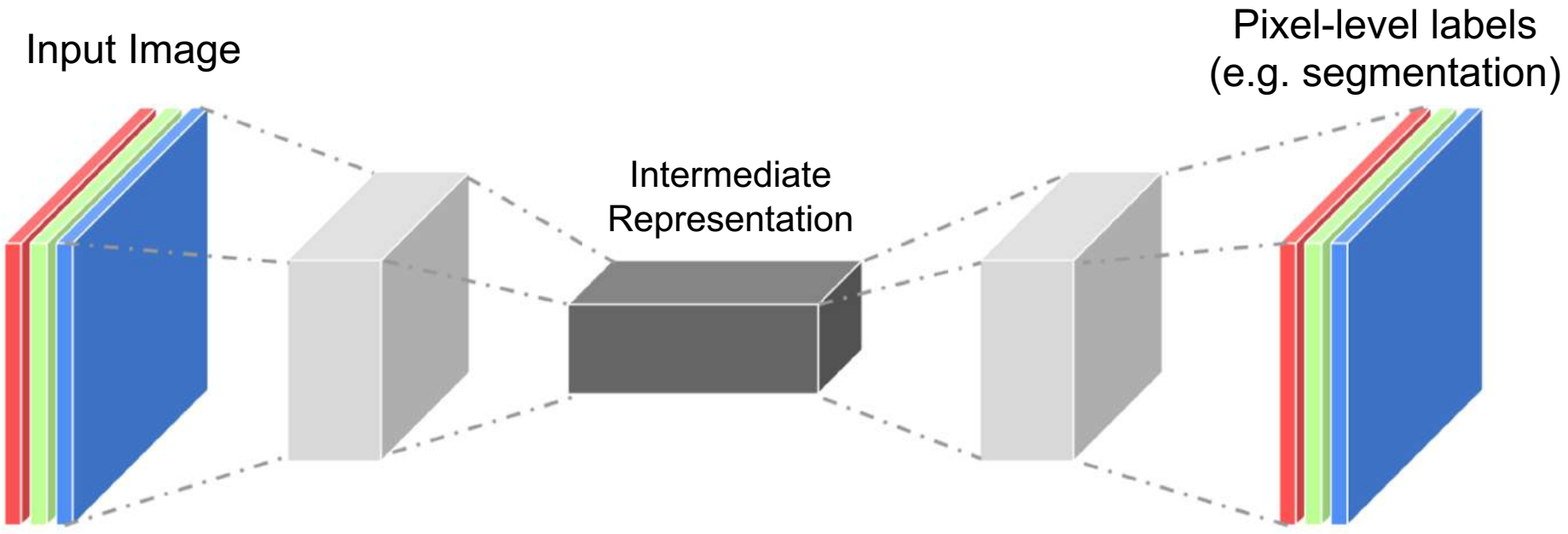
dX

(D_{in} x N x N) x #Patches

Transposed Convolution (Deconvolution) Layer



Transposed Convolution (Deconvolution) Layer

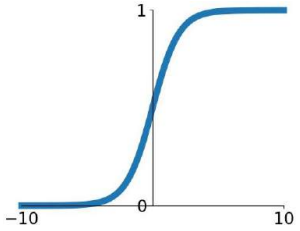


- Often used to up-sample the feature map (subsume bilinear interpolation)
- Sometimes also known as deconvolution, fractionally strided convolutions

Nonlinear Activation Layer

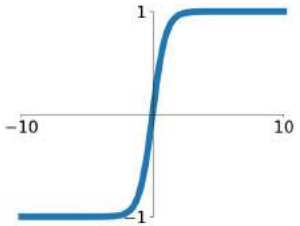
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



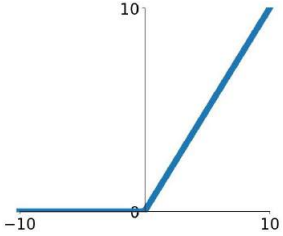
tanh

$$\tanh(x)$$



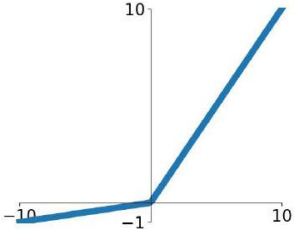
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

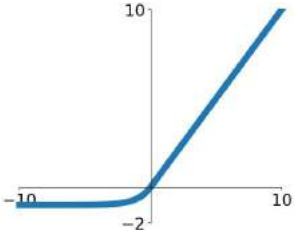


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

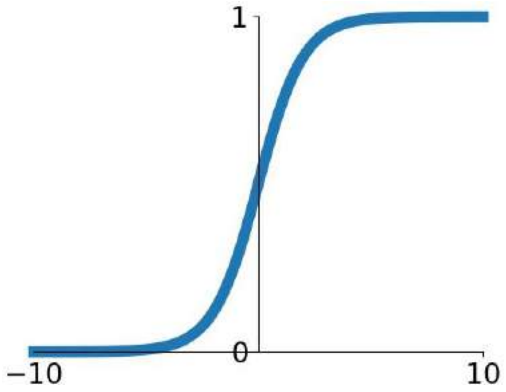
ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



All these functions are “differentiable”

Nonlinear Activation Layer

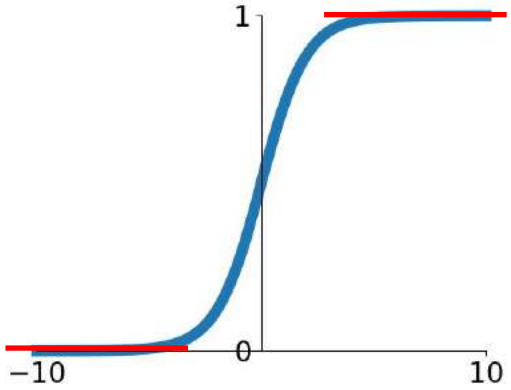


Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

Nonlinear Activation Layer



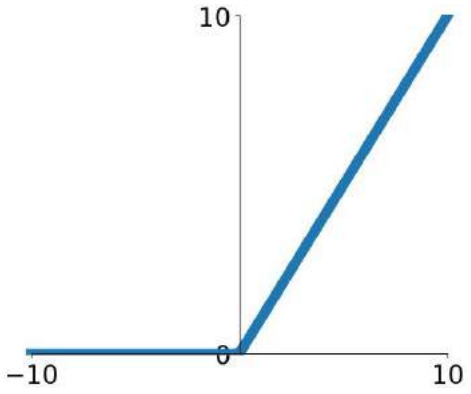
Sigmoid

$$\sigma(x) = 1/(1 + e^{-x})$$

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron

- Saturated neurons "kill" the gradients
- Exponential function is expensive

Nonlinear Activation Layer



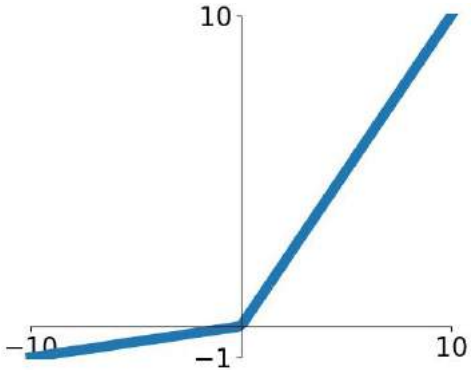
ReLU
(Rectified Linear Unit)

f(x) = max(0, x)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Gradient knockout (never activate -> never update)

Nonlinear Activation Layer



- Does not saturate (in +region)
- Computationally efficient
- Converges much faster than sigmoid in practice (e.g. 6x)
- Will no “die”

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PreLU)

$$f(x) = \max(\alpha x, x)$$

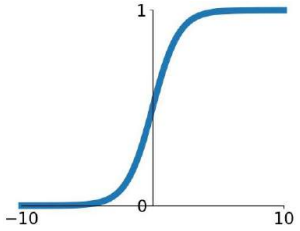
backprop into \alpha (parameter)

[Mass et al., 2013]
[He et al., 2015]

Nonlinear Activation Layer

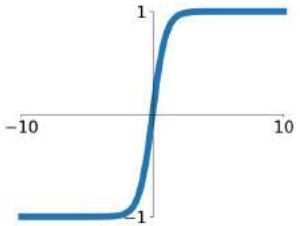
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



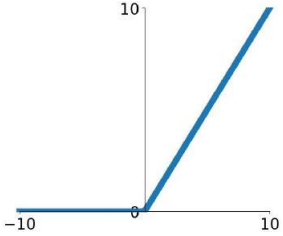
tanh

$$\tanh(x)$$



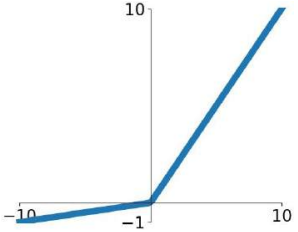
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

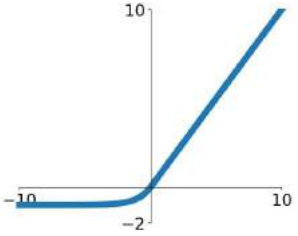


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

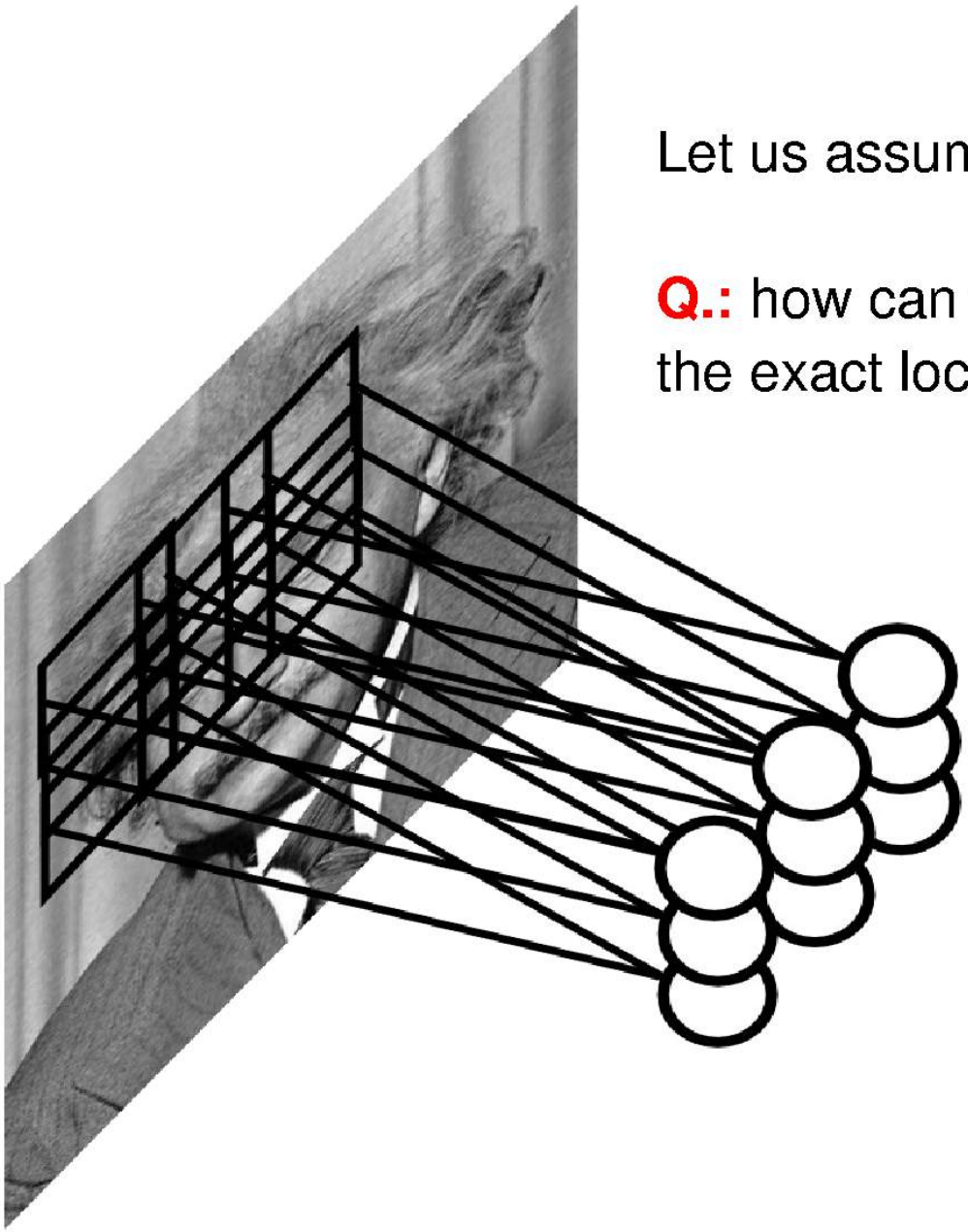
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



ReLU is often used by default

Try out Leaky ReLU, Maxout, ELU …

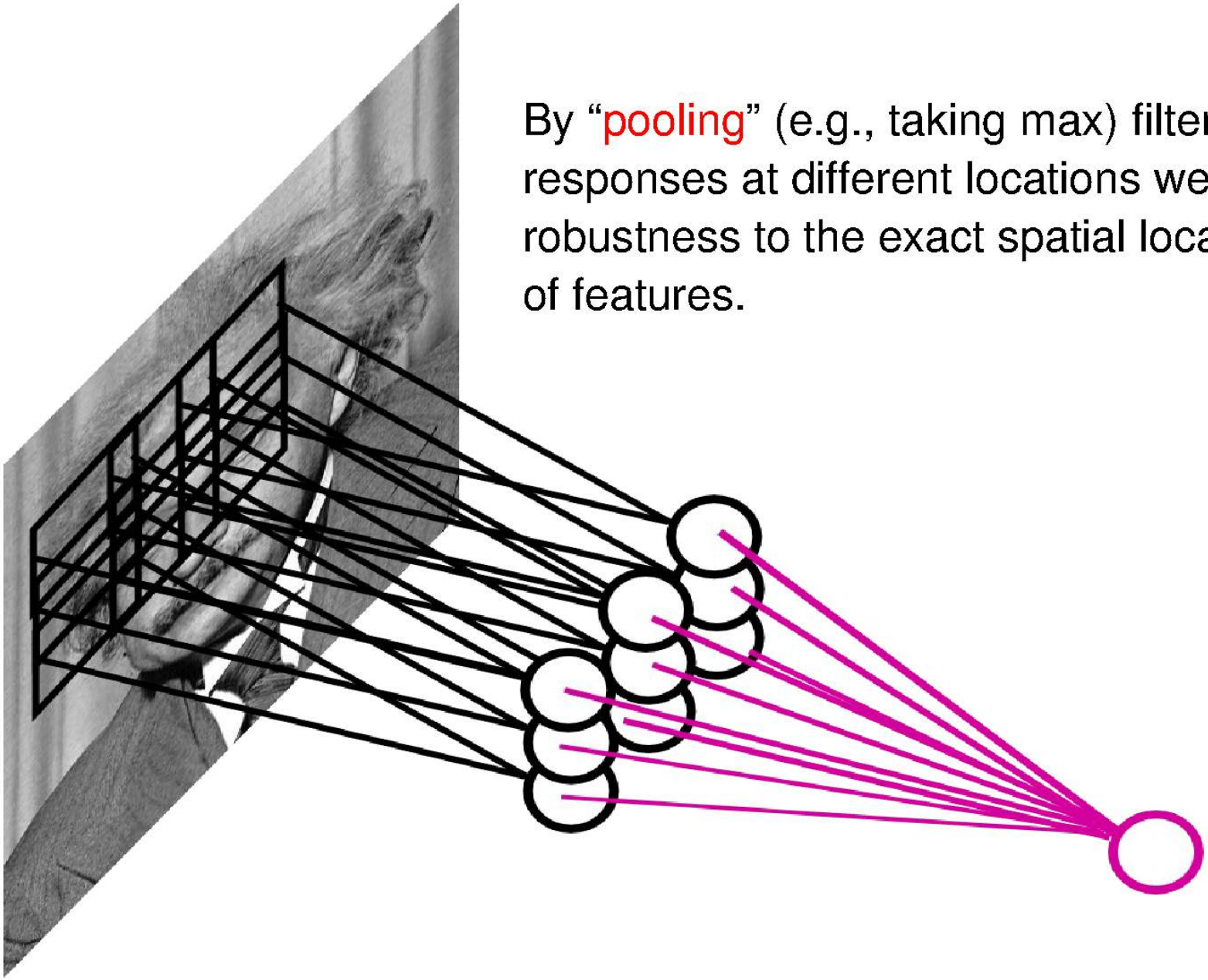
Pooling Layer



Let us assume filter is an “eye” detector.

Q.: how can we make the detection robust to the exact location of the eye?

Pooling Layer



By "pooling" (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

Pooling Layer: Examples

Max-pooling:

$$h_j^n(x,y) = max_{\bar{x}\in N(x),\, \bar{y}\in N(y)} h_j^{n-1}(\bar{x},\bar{y})$$

Average-pooling:

$$h_j^n(x,y) = 1/K \sum_{\bar{x}\in N(x),\, \bar{y}\in N(y)} h_j^{n-1}(\bar{x},\bar{y})$$

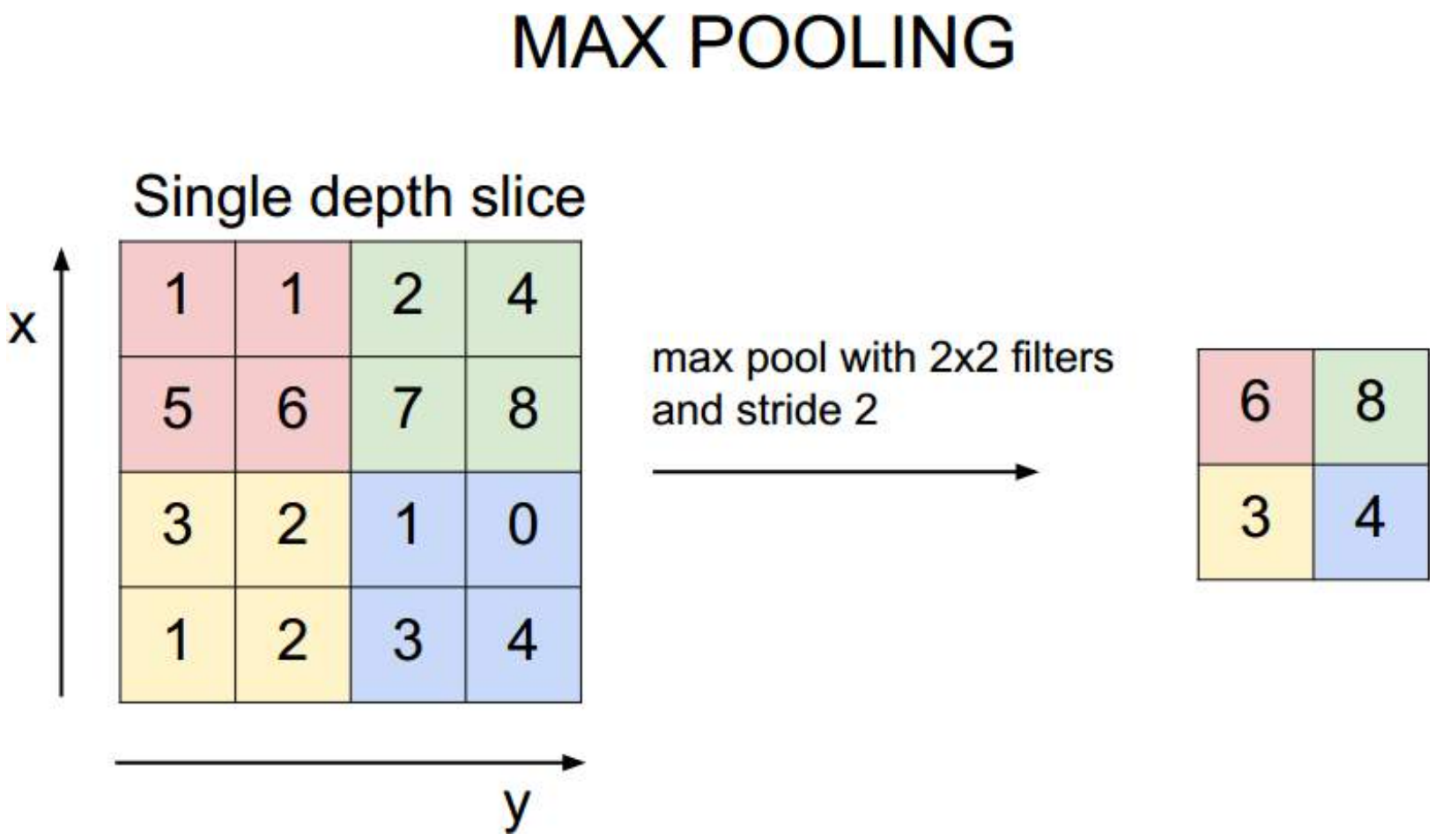
L2-pooling:

$$h_j^n(x,y) = \sqrt{\sum_{\bar{x}\in N(x),\, \bar{y}\in N(y)} h_j^{n-1}(\bar{x},\bar{y})^2}$$

L2-pooling over features:

$$h_j^n(x,y) = \sqrt{\sum_{k\in N(j)} h_k^{n-1}(x,y)^2}$$

Pooling Layer: Examples



Question: How to compute the gradients?

Hint: Book keeping (save the index in F-Prop)

Pooling Layer

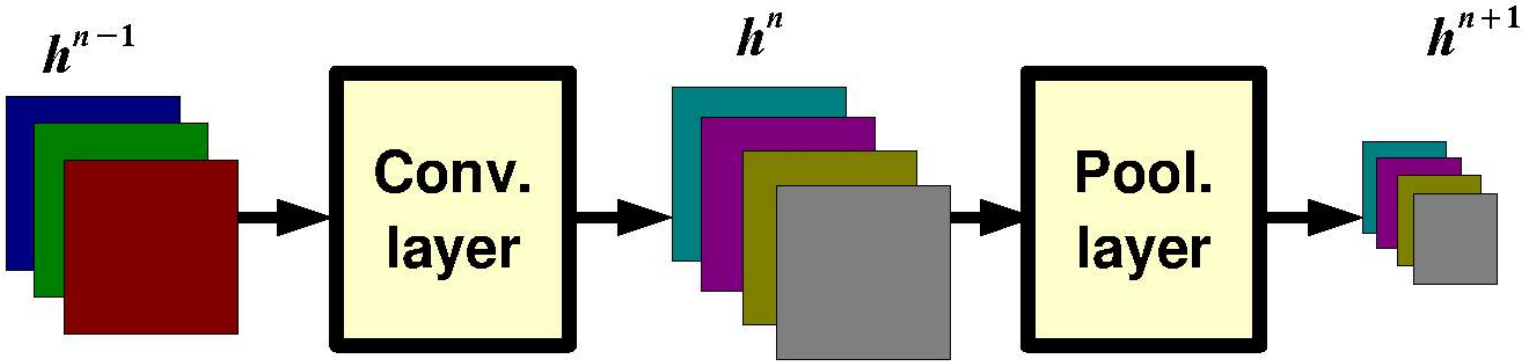
Question: What is the size of the output?

Hint: Think about convolutions!

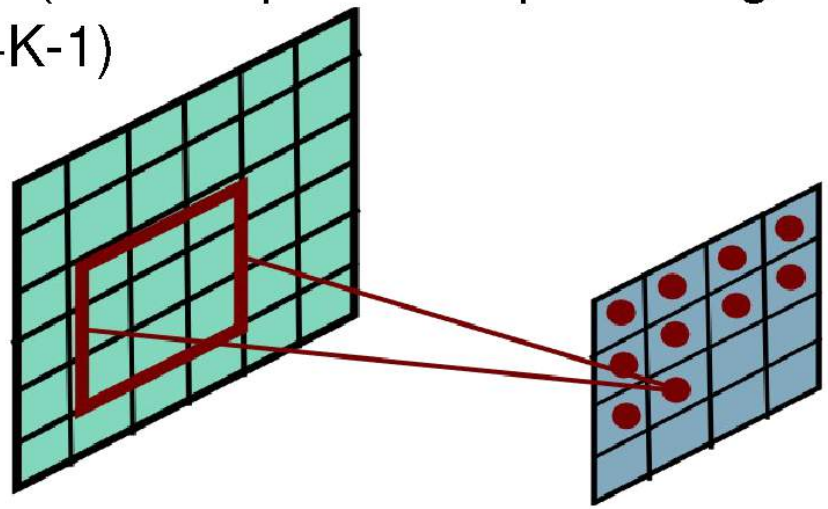
Question: How should I set the size of the pools?

Answer: It depends on how much “invariant” or robust to distortions we want the representation to be. It is best to pool slowly (via a few stacks of conv-pooling layers).

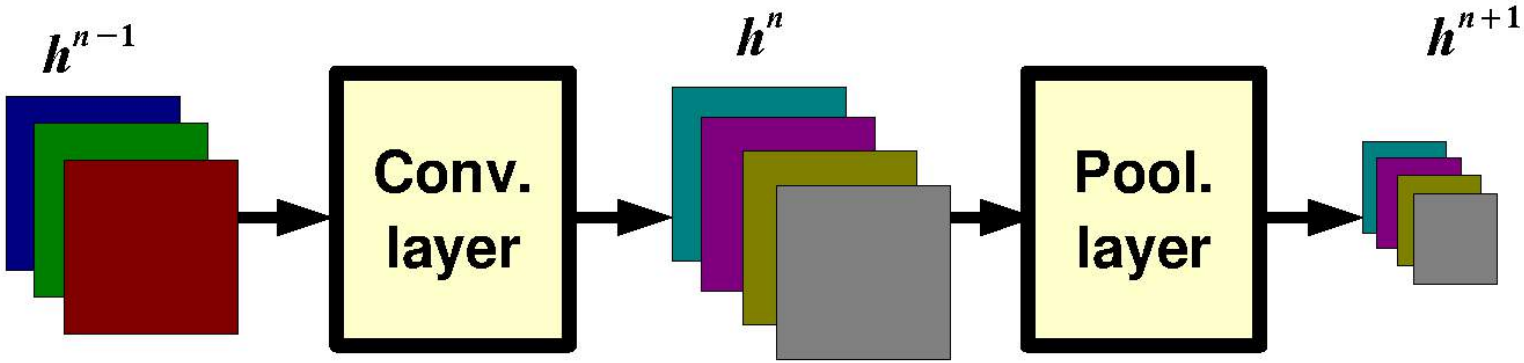
Pooling Layer: Receptive Field Size



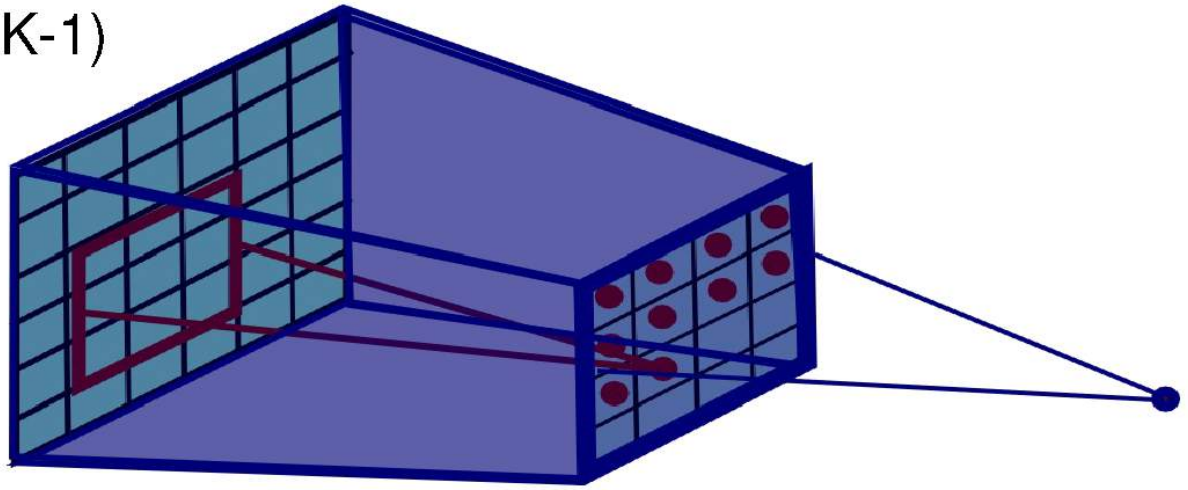
If convolutional filters have size KxK and stride 1, and pooling layer has pools of size PxP, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: (P+K-1)x(P+K-1)



Pooling Layer: Receptive Field Size

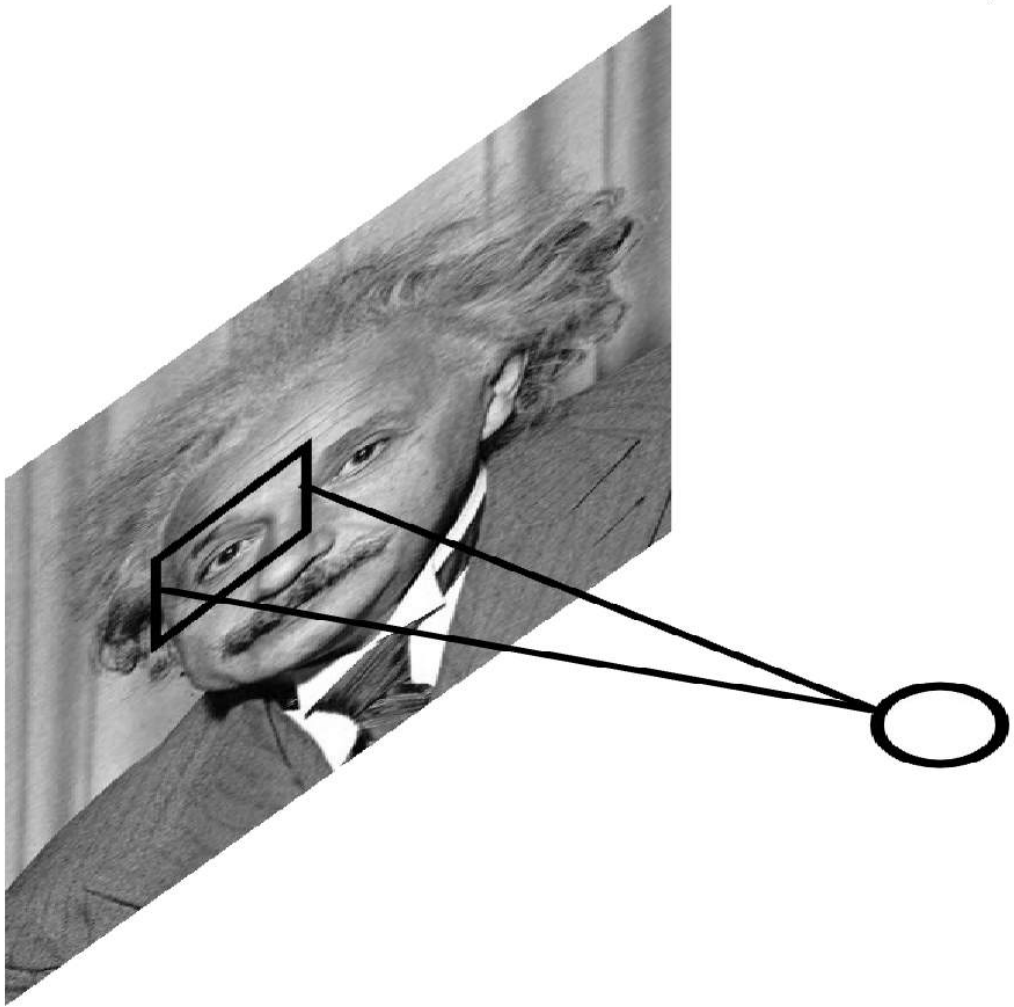


If convolutional filters have size KxK and stride 1, and pooling layer has pools of size PxP, then each unit in the pooling layer depends upon a patch (at the input of the preceding conv. layer) of size: (P+K-1)x(P+K-1)



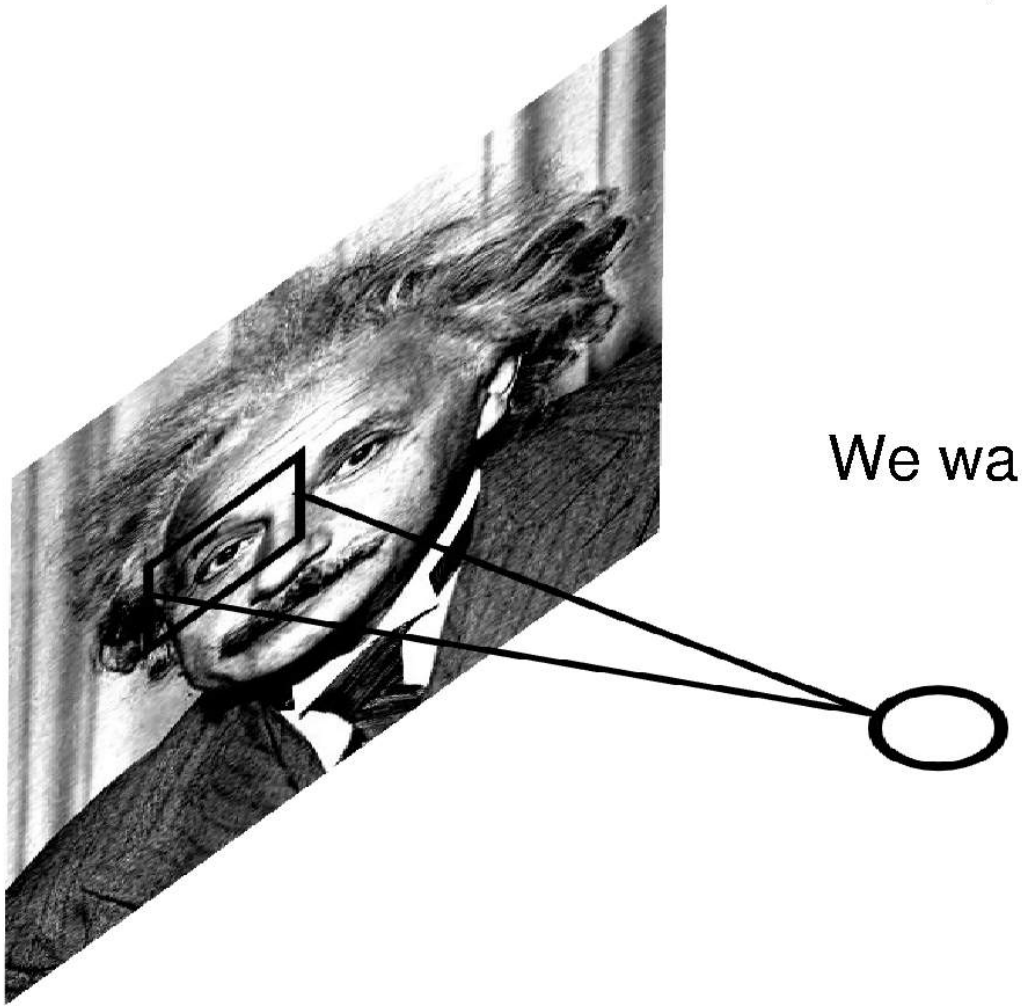
Local Contrast Normalization

$$h^{i+1}(x,y)=\frac{h^{i}(x,y)-m^{i}(N(x,y))}{\sigma^{i}(N(x,y))}$$



Local Contrast Normalization

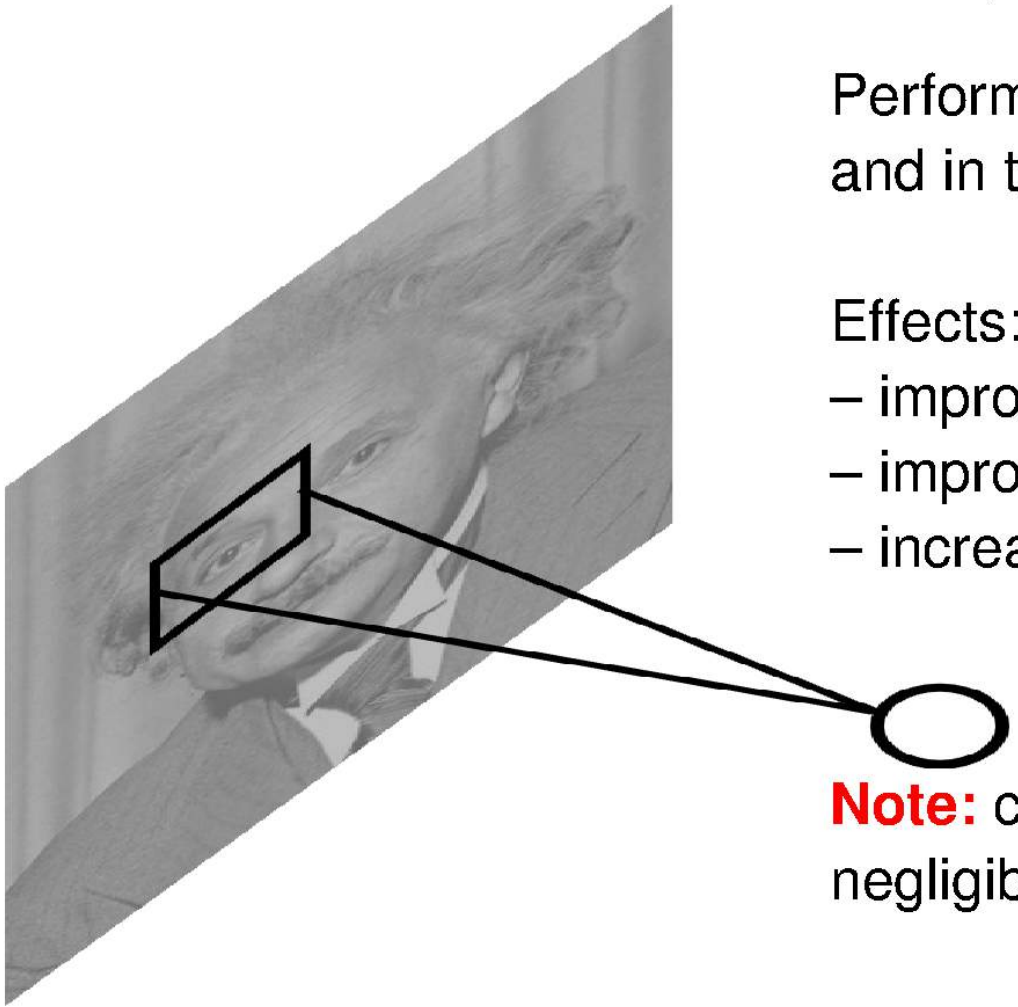
$$h^{i+1}(x,y)=\frac{h^{i}(x,y)-m^{i}(N(x,y))}{\sigma^{i}(N(x,y))}$$



We want the same response.

Local Contrast Normalization

$$h^{i+1}(x,y)=\frac{h^{i}(x,y)-m^{i}(N(x,y))}{\sigma^{i}(N(x,y))}$$



Performed also across features and in the higher layers..

Effects:
- improves invariance
- improves optimization
- increases sparsity

Note: computational cost is negligible w.r.t. conv. layer.

Batch Normalization Layer

"Batch norm is a widely adopted technique that enables faster and more stable training of deep neural networks"

Consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

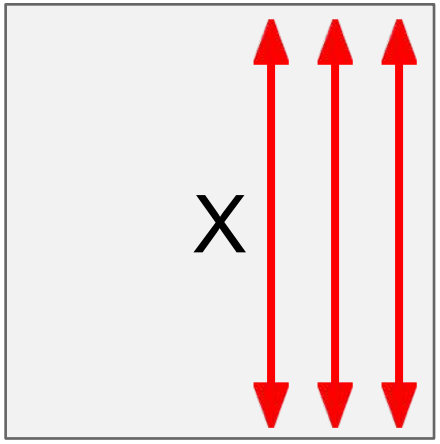
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

[Ioffe and Szegedy, 2015]

Batch Normalization Layer

1D Case: N*D*1*1 (N = #samples, D = #feat dims)



1. compute the empirical mean and variance independently for each dimension.

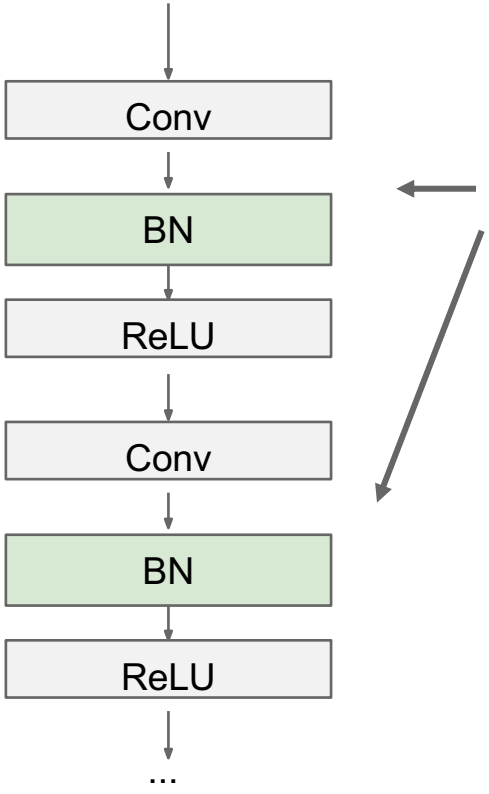
2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

2D Case: N*D*H*W (HxW = Spatial resolution)

- Spatial Batchnorm: average across HxW

Batch Normalization Layer



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

- Need a sufficiently large batch size (to aggregate stats within mini-batch)

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization Layer

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization Layer

Training

- Take a batch, compute mean and variance
- Use the mean and variance to get new values
- Use the new values to feed to activation layer

Test

- Compute mean and variance during training

mean = alpha*mean + (1- alpha)*batch_mean

- Use the computed mean and variance at test time.

Batch Normalization Layer

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

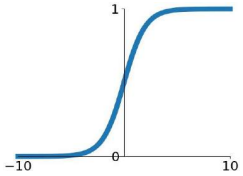
Exact reasons for BatchNorm's effectiveness are poorly understood ☹

Internal covariate shift? Smooth the optimization landscape?

[Ioffe and Szegedy, 2015], [Santurkar, Tsipras, Ilyas, Madry, 2018]

Output Normalization Layer

Sigmoid



$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- Normalize the outputs into the range of [0,1]
- Can be considered as a probability distribution for a binary random variable
- Does not have any parameters

Output Normalization Layer

Sigmoid

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- Normalize the outputs into the range of [0,1]
- Can be considered as a probability distribution for a binary random variable
- Does not have any parameters

Softmax

$$softmax(x^k) = \frac{\exp(x^k)}{\sum_j \exp(x^j)}$$

- Normalize the outputs into the range of [0,1]
- A probability distribution for a multinomial distribution (*labels are mutually exclusive*)
- Does not have any parameters

Question: Unstable gradients from the exponential function?

Answer: Merge the normalization into loss function during training

Loss Function

Many choices depending on the task

Cross Entropy Loss $H(y,p) = -\sum_j y_j \log(p_j)$

- Used for classification (assuming mutually exclusive labels)

L1/2 Loss $L_{1/2}(y,\hat{y}) = ||y - \hat{y}||_{1/2}$

- Used for regression

KL Divergence $KL(p,q) = \sum_j p_j \log(\frac{p_j}{q_j})$

- Matching the output to a reference distribution *q*

Hinge Loss $L_{hinge} = \max(0,1 - y\hat{y})$

- Approximate the indicator function, used by SVM

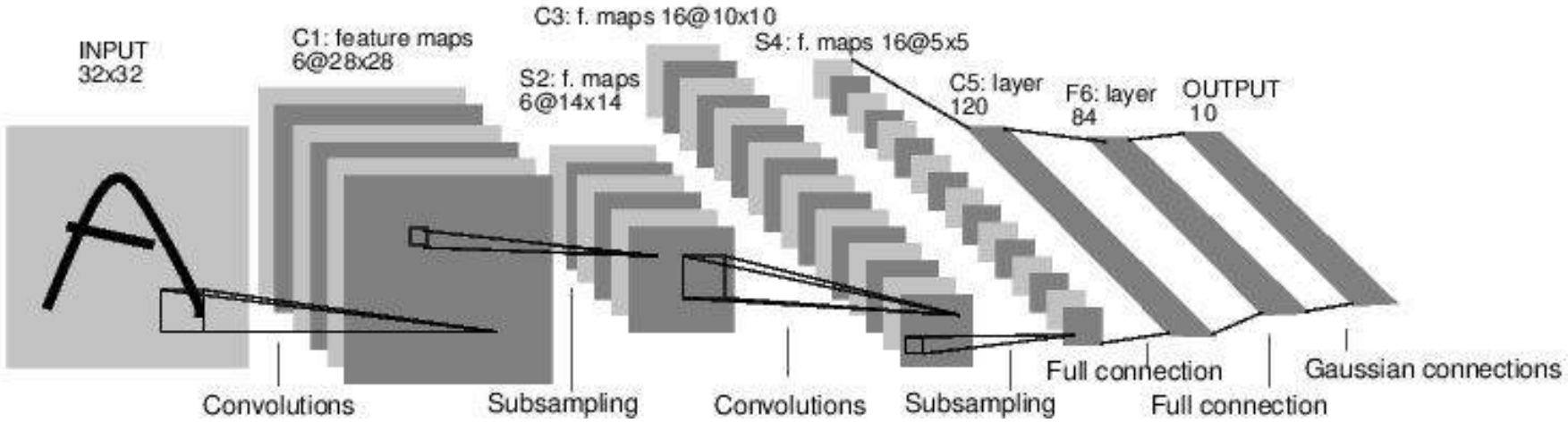
Again, all these functions are "differentiable"

Convolutional Neural Networks

ConvNets: Case Study

Case Study: LeNet-5

[LeCun et al., 1998]

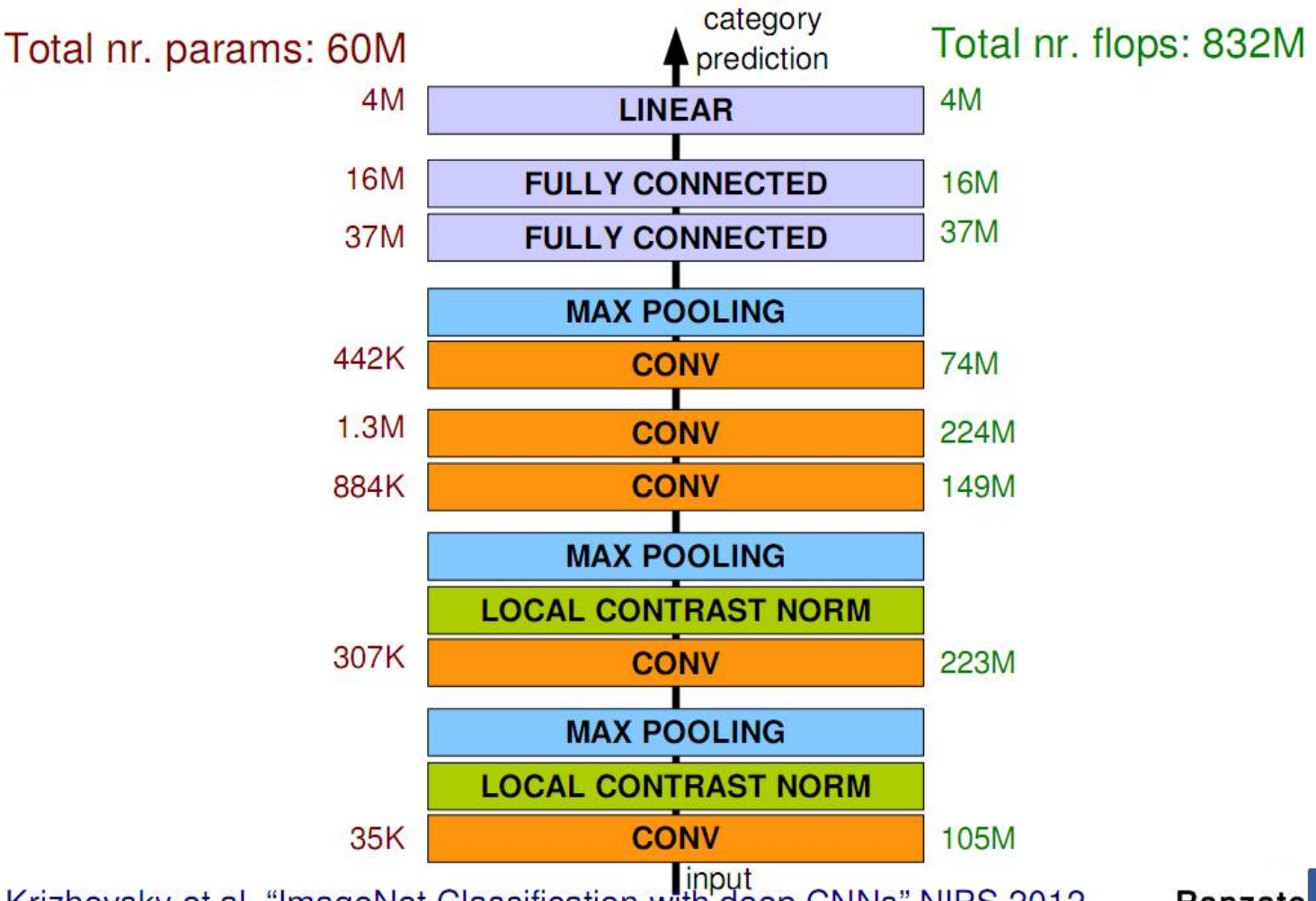


Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

ConvNets: Case Study

Imagenet Network

Architecture for Classification

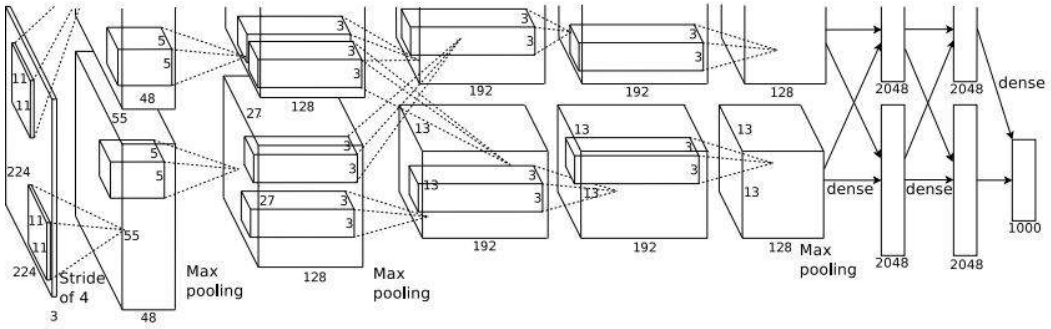


Krizhevsky et al. "ImageNet Classification with deep CNNs" NIPS 2012

ConvNets: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

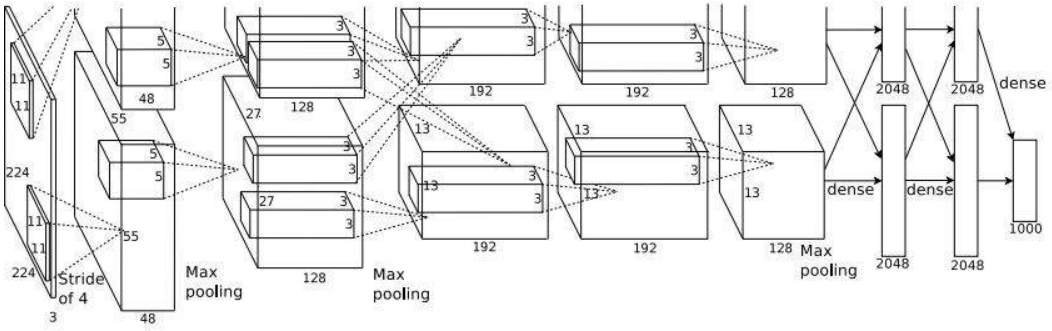
=>

Q: what is the output volume size? Hint: (227-11)/4+1 = 55

ConvNets: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

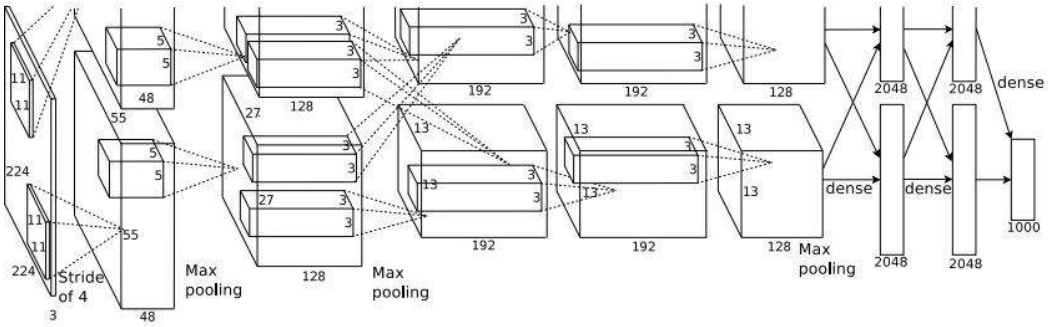
First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**

Q: What is the total number of parameters in this layer?

ConvNets: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]



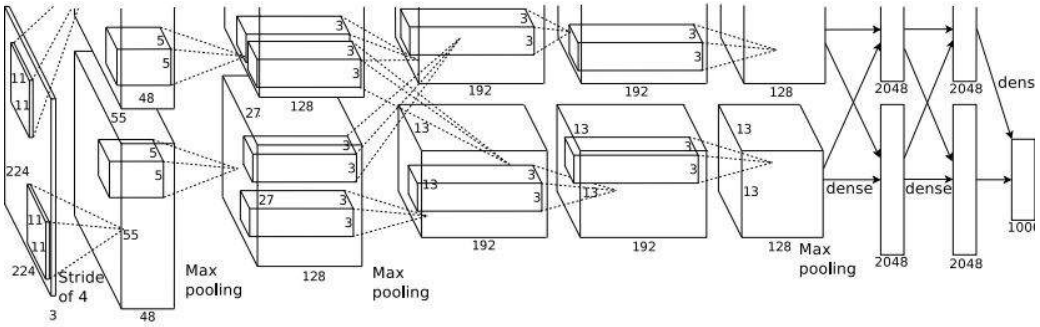
Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4
=>
Output volume **[55x55x96]**
Parameters: (11*11*3)*96 = **35K**

ConvNets: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images
After CONV1: 55x55x96

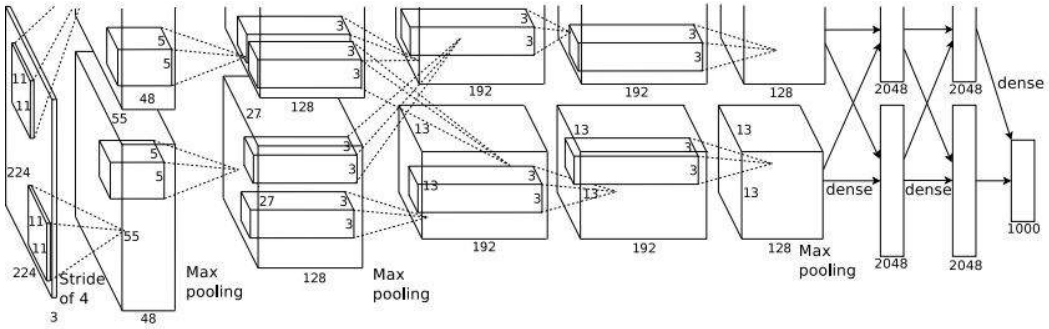
Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: (55-3)/2+1 = 27

ConvNets: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images
After CONV1: 55x55x96

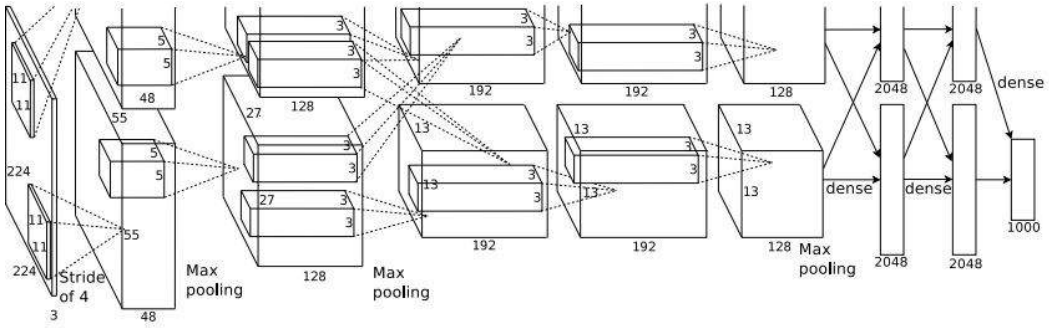
Second layer (POOL1): 3x3 filters applied at stride 2
Output volume: 27x27x96

Q: what is the number of parameters in this layer?

ConvNets: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]

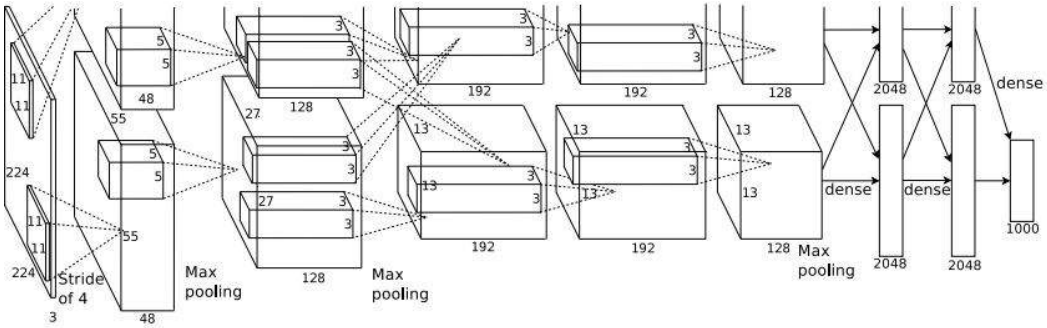


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

ConvNets: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]



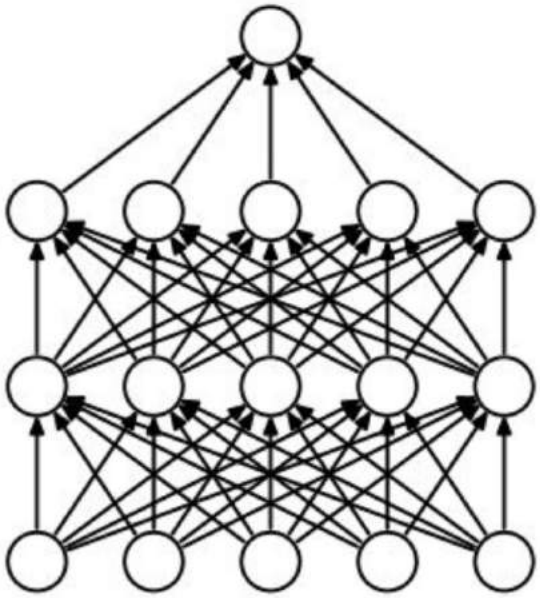
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons (w. Dropout)
[4096] FC7: 4096 neurons (w. Dropout)
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
-first use of ReLU
-used Contrast Norm (not common anymore)
-heavy data augmentation
-dropout 0.5
-batch size 128
-SGD Momentum 0.9
-Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
-L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

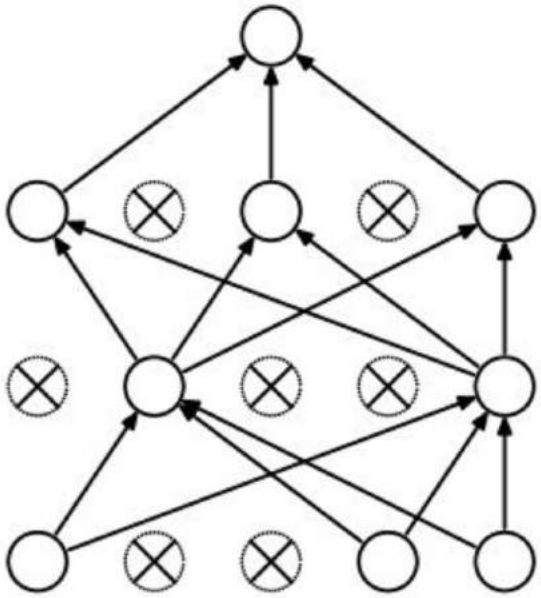
ConvNets: Regularization

Regularization: **Dropout**

"randomly set some neurons to zero in the forward pass"



(a) Standard Neural Net

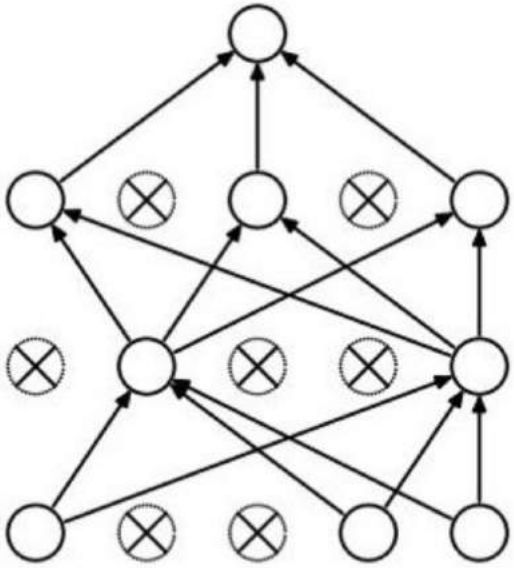


(b) After applying dropout.

[Srivastava et al., 2014]

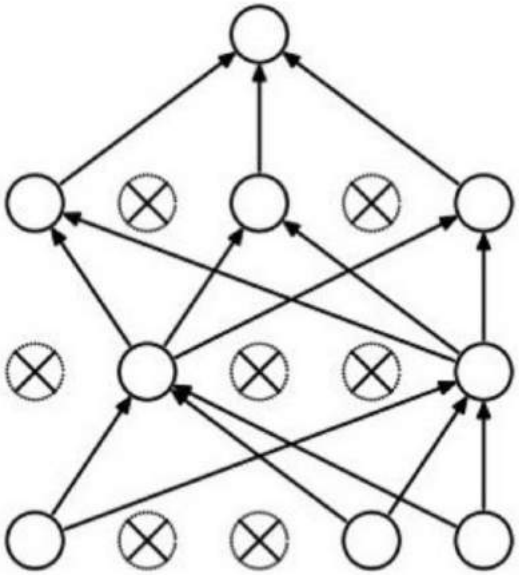
ConvNets: Regularization

Waaaait a second…
How could this possibly be a good idea?



ConvNets: Regularization

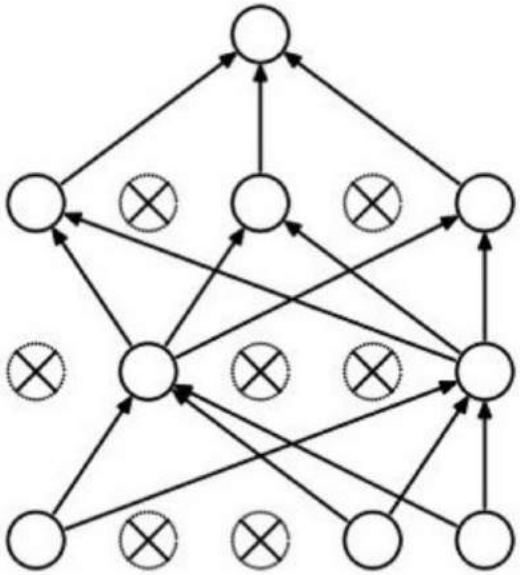
Waaaait a second…
How could this possibly be a good idea?



Forces the network to have a redundant representation.

ConvNets: Regularization

Waaaait a second…
How could this possibly be a good idea?



Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained on only ~one datapoint.

Don’t add Dropout to convolutional layers!
(unless you have a large batch size)

ConvNets: Case Study

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

best model

11.2% top 5 error in ILSVRC 2013
->
7.3% top 5 error

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

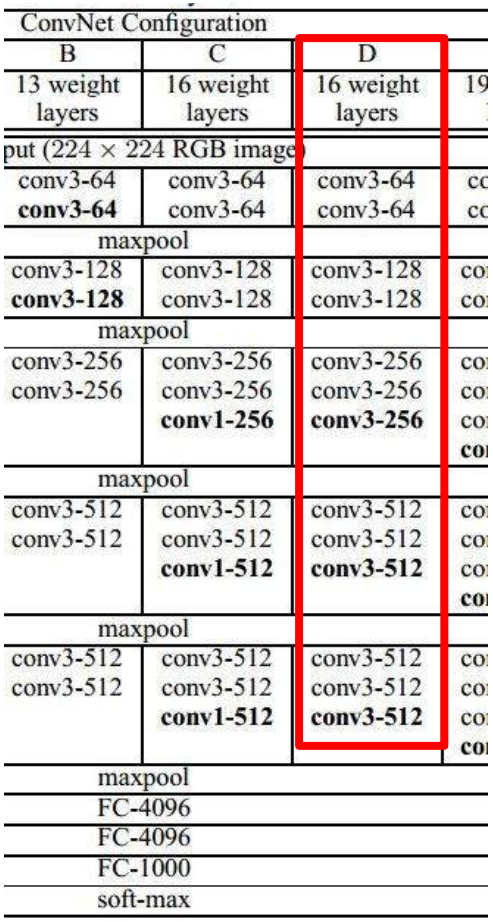
Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

ConvNets: Case Study

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000



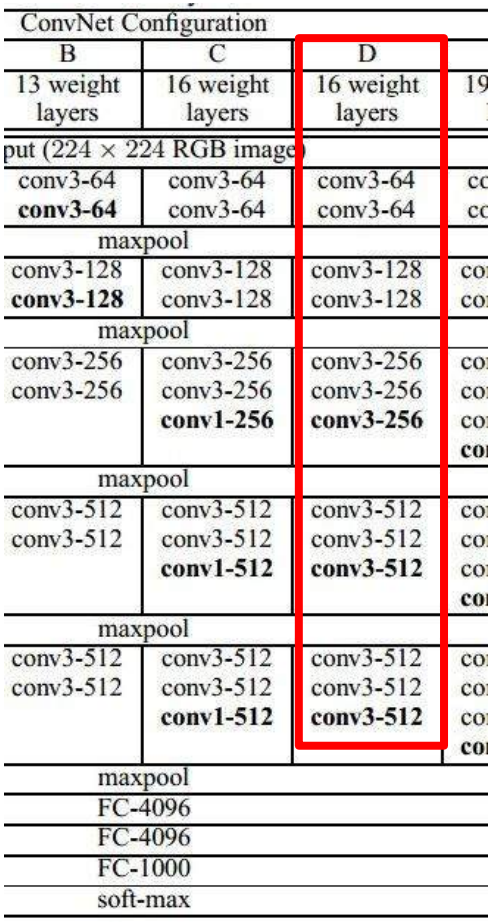
ConvNet Configuration			
B	C	D	
13 weight layers	16 weight layers	16 weight layers	19
put (224 × 224 RGB image)			
conv3-64	conv3-64	conv3-64	co
conv3-64	conv3-64	conv3-64	co
maxpool			
conv3-128	conv3-128	conv3-128	co
conv3-128	conv3-128	conv3-128	co
maxpool			
conv3-256	conv3-256	conv3-256	co
conv3-256	conv3-256	conv3-256	co
	conv1-256	**conv3-256**	co
			co
maxpool			
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	**conv3-512**	co
			co
maxpool			
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	**conv3-512**	co
			co
maxpool			
FC-4096			
FC-4096			
FC-1000			
soft-max			

ConvNets: Case Study

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

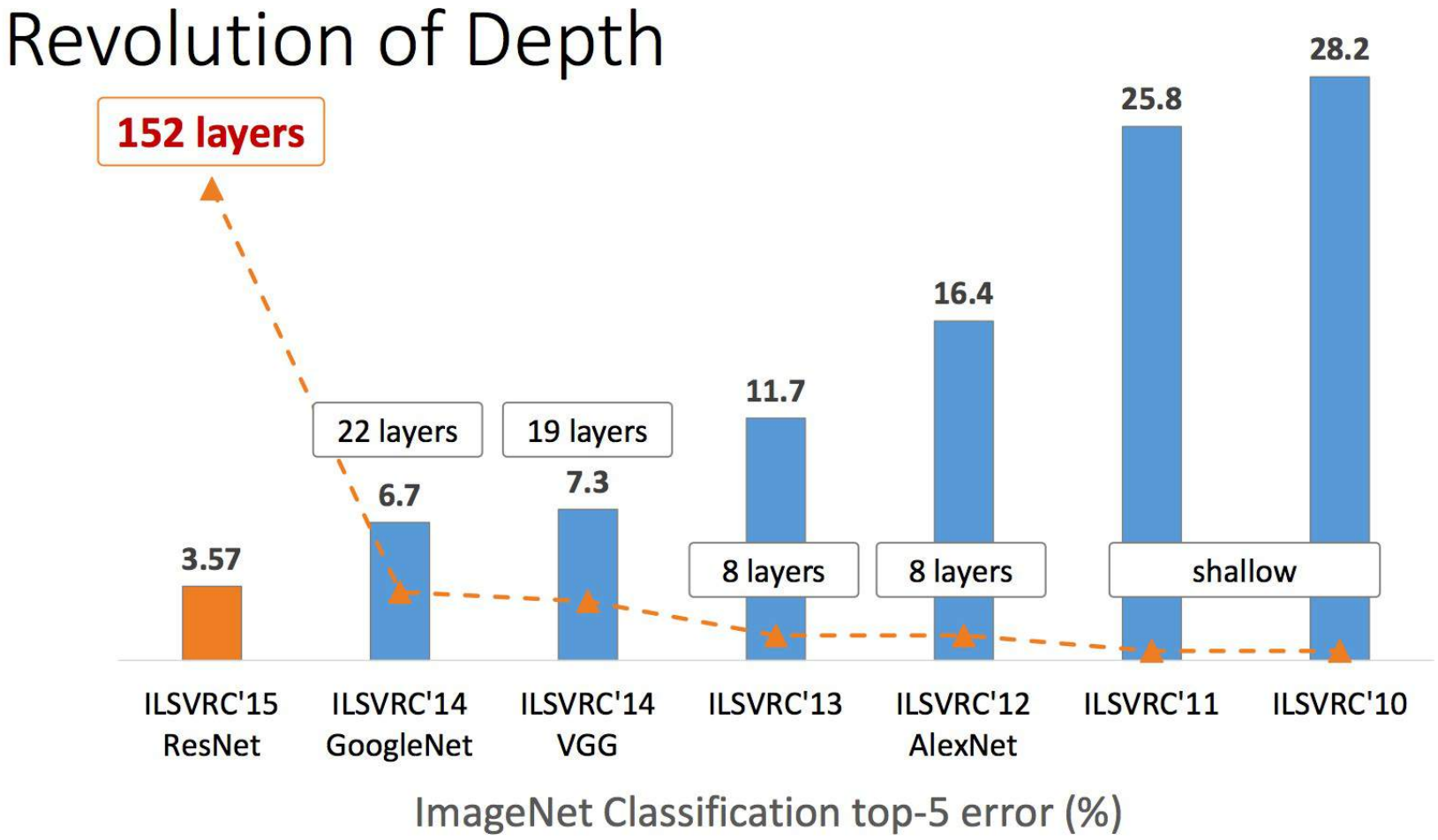


ConvNet Configuration			
B	C	D	
13 weight layers	16 weight layers	16 weight layers	19
put (224 × 224 RGB image)			
conv3-64	conv3-64	conv3-64	co
conv3-64	conv3-64	conv3-64	co
maxpool			
conv3-128	conv3-128	conv3-128	co
conv3-128	conv3-128	conv3-128	co
maxpool			
conv3-256	conv3-256	conv3-256	co
conv3-256	conv3-256	conv3-256	co
	conv1-256	**conv3-256**	co
			co
maxpool			
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	**conv3-512**	co
			co
maxpool			
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	**conv3-512**	co
			co
maxpool			
FC-4096			
FC-4096			
FC-1000			
soft-max			

ConvNets: Important Points

- Basic unit = $[\text{Conv} + (\text{BN}) + \text{ReLU}]_{xN}$ + Pooling
- Slow increase of receptive fields
- Strong regularization via
  - Data augmentation (prevent overfitting)
  - Batch normalization (how?)
  - Weight decay (L2 regularization on weights)
  - SGD (noisy gradients help the learning!)
  - Dropout (ensemble many models)

ConvNets: Depth

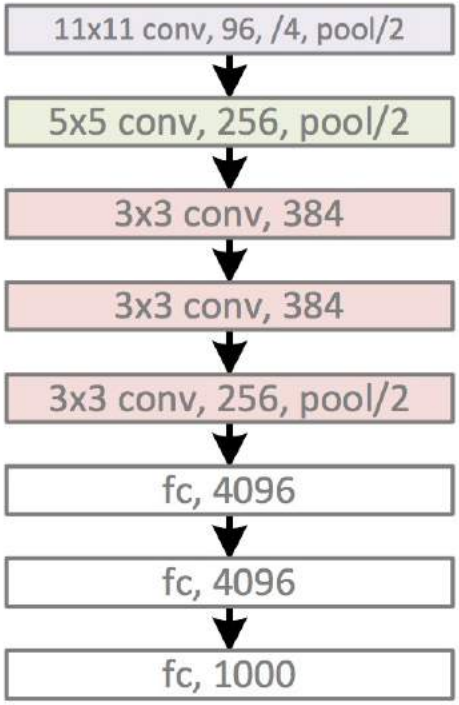


Slide Credit: Kaiming He

ConvNets: Depth

Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)



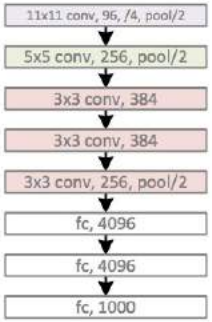
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

Slide Credit: Kaiming He

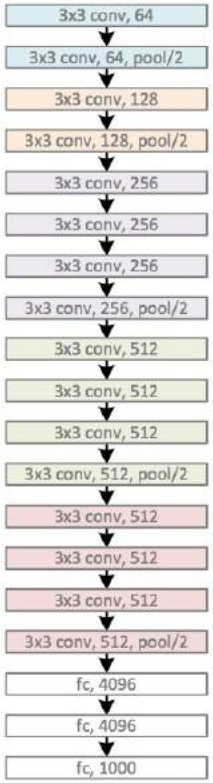
ConvNets: Depth

Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)



VGG, 19 layers
(ILSVRC 2014)



GoogleNet, 22 layers
(ILSVRC 2014)

Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

Slide Credit: Kaiming He

ConvNets: Depth

Revolution of Depth

AlexNet, 8 layers
(ILSVRC 2012)

VGG, 19 layers
(ILSVRC 2014)

ResNet, 152 layers
(ILSVRC 2015)

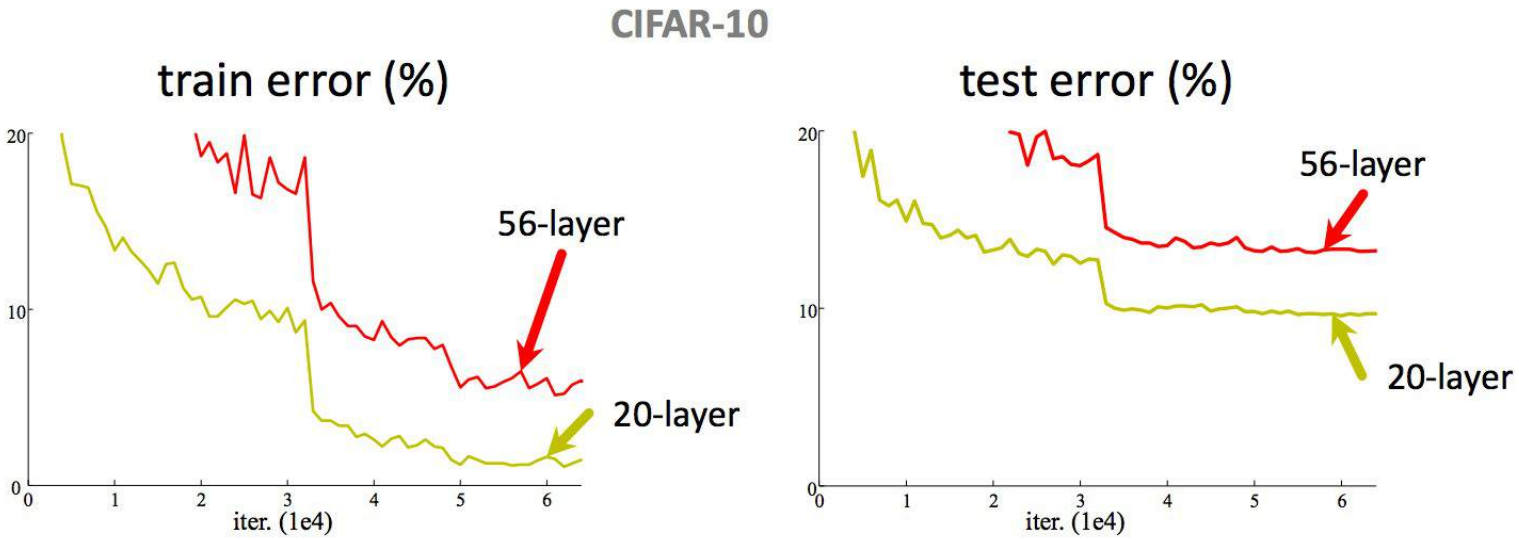
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

Slide Credit: Kaiming He

Is learning better networks as simple as stacking more layers?

ConvNets: Depth

Simply stacking layers?

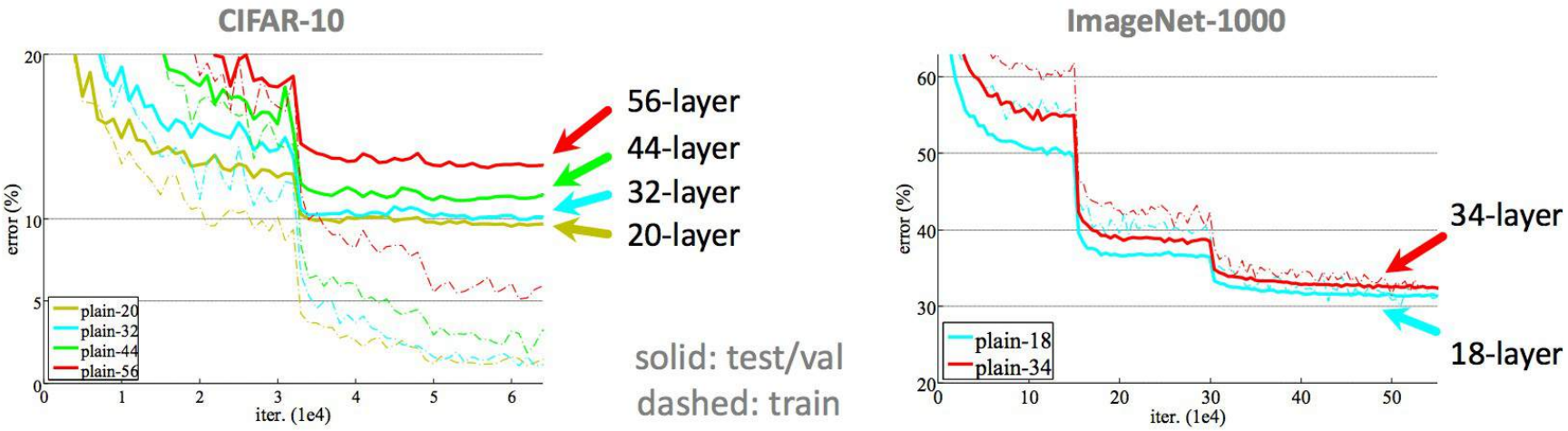


- *Plain* nets: stacking 3x3 conv layers…
- 56-layer net has **higher training error** and test error than 20-layer net

Slide Credit: Kaiming He

ConvNets: Depth

Simply stacking layers?

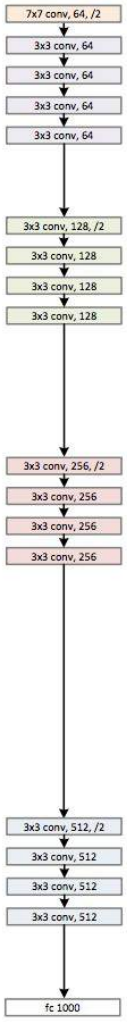


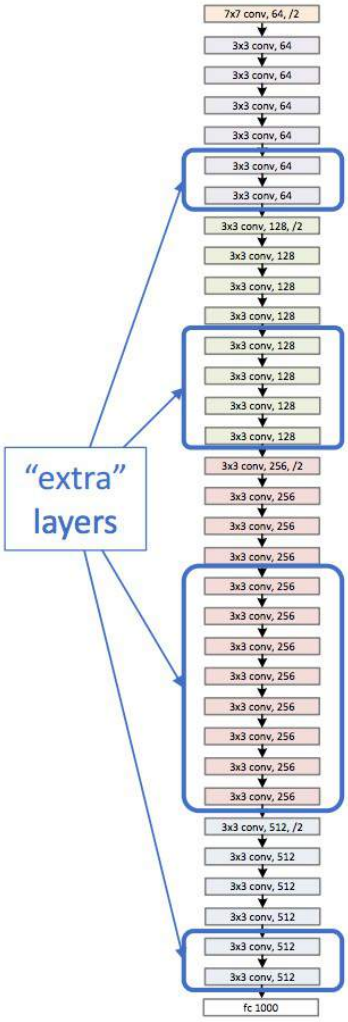
- "Overly deep" plain nets have **higher training error**
- A general phenomenon, observed in many datasets

Slide Credit: Kaiming He

ConvNets: ResNet

a shallower model (18 layers)





a deeper counterpart (34 layers)

- A deeper model should not have **higher training error**
- A solution *by construction*:
  - original layers: copied from a learned shallower model
  - extra layers: set as identity
  - at least the same training error
- Optimization difficulties: solvers cannot find the solution when going deeper… ***e.g., Gradient vanishing?***

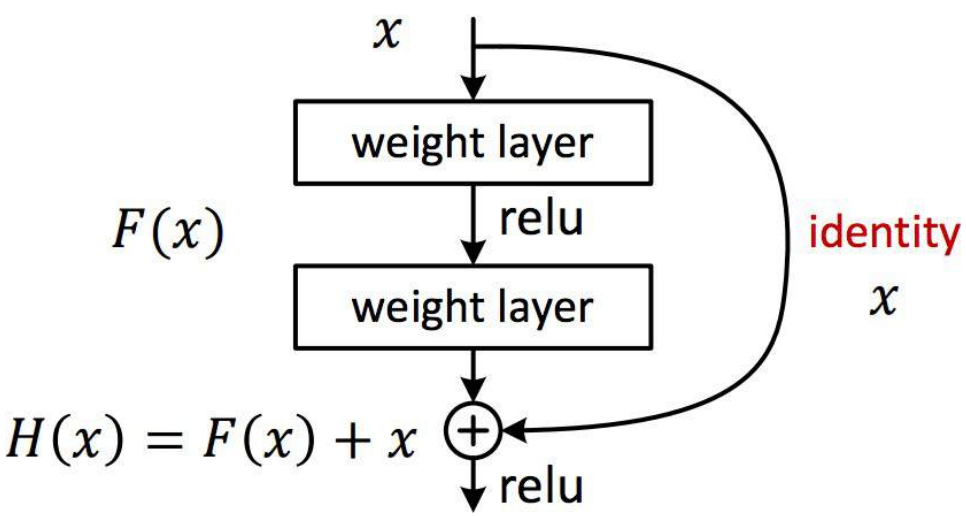
Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. "Deep Residual Learning for Image Recognition". arXiv 2015.

Slide Credit: Kaiming He

ConvNets: ResNet

Deep Residual Learning

- Residual net



$H(x)$ is any desired mapping,

~~hope the 2 weight layers fit $H(x)$~~

hope the 2 weight layers fit $F(x)$

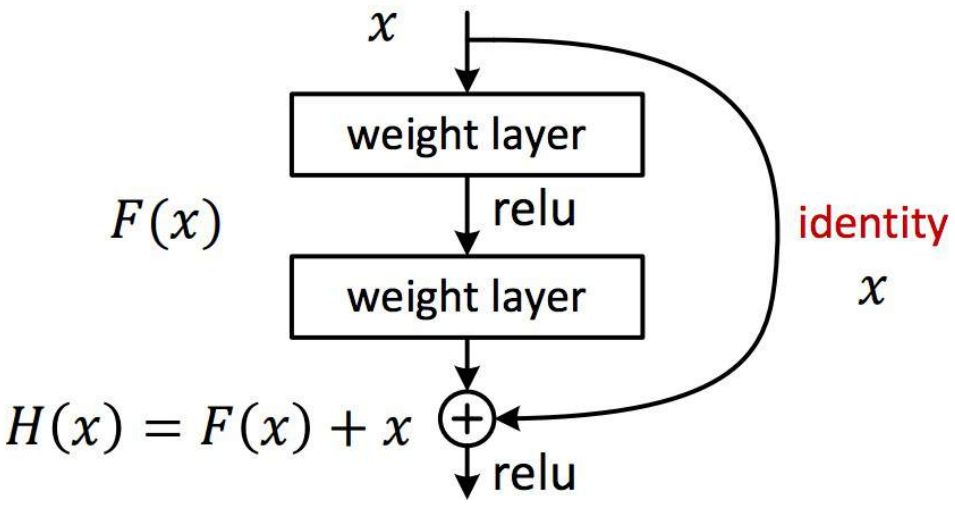
let $H(x) = F(x) + x$

Slide Credit: Kaiming He

ConvNets: ResNet

Deep Residual Learning

- $F(x)$ is a residual mapping w.r.t. identity



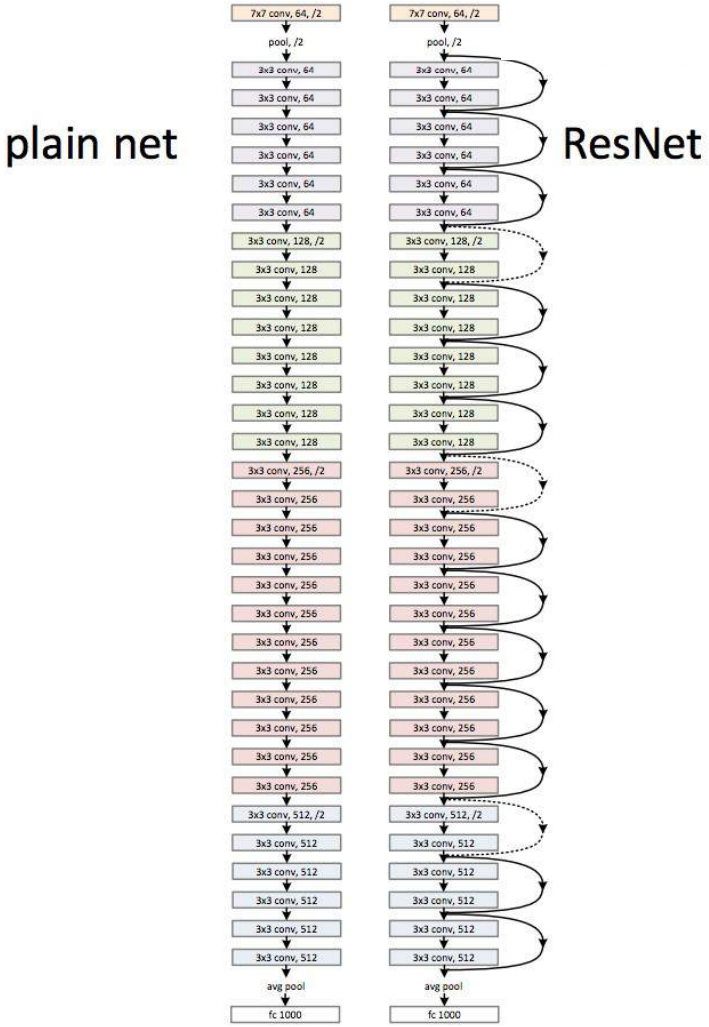
- If identity were optimal, easy to set weights as 0
- If optimal mapping is closer to identity, easier to find small fluctuations

Slide Credit: Kaiming He

ConvNets: ResNet

Network "Design"

- Keep it simple
- Our basic design (VGG-style)
  - all 3x3 conv (almost)
  - spatial size /2 => # filters x2
  - Simple design; just deep!
- Other remarks:
  - no max pooling (almost)
  - no hidden fc
  - no dropout

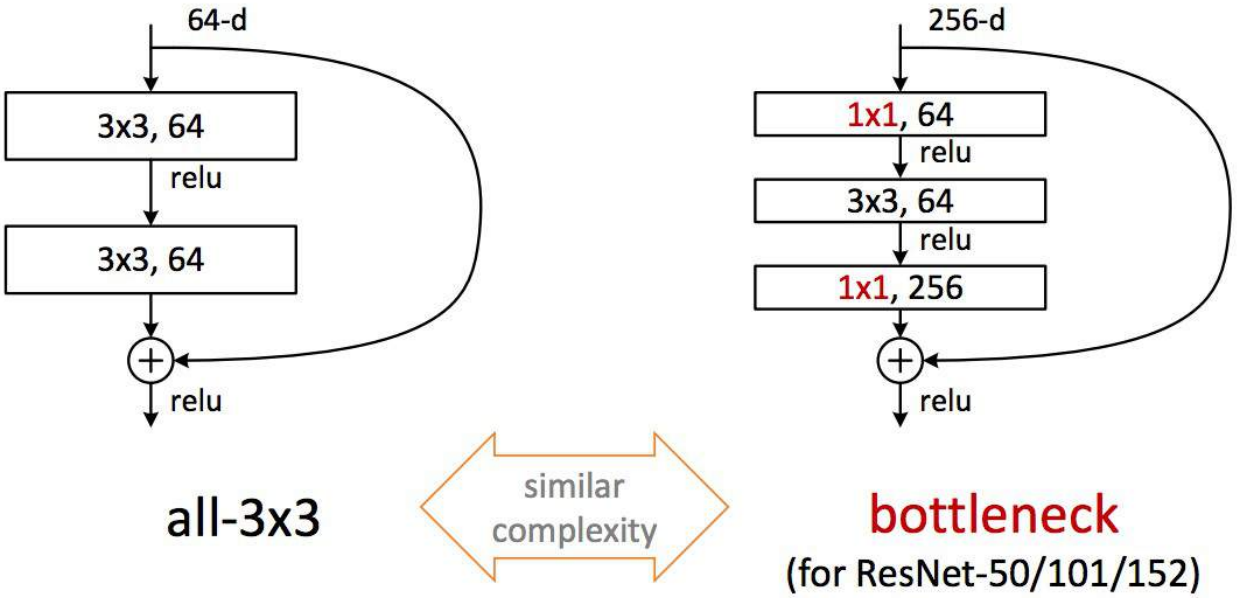


Slide Credit: Kaiming He

ConvNets: ResNet

ImageNet experiments

- A practical design of going deeper



Slide Credit: Kaiming He

ConvNets: ResNet

- Pre-conditioned better
- No more vanishing gradients
- Ensemble effect (aggregate many paths for the prediction)
- Strong performance on a number of tasks!

[Veit, Wilber, Belongie]

Training Convolutional Neural Networks

ConvNets: Training

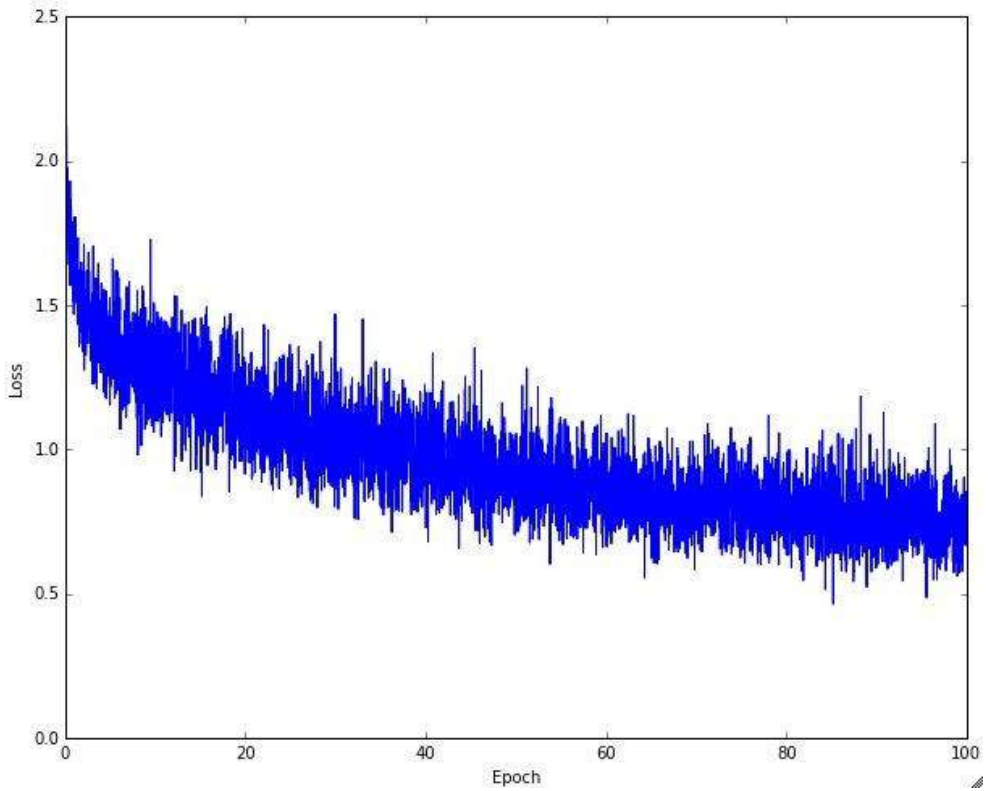
Initialize all network parameters

Loop:

- **Sample** a batch of data
- **Forward** prop it through the graph (network), get loss
- **Backprop** to calculate the gradients
- **Update** the parameters using the gradient

Till convergence …

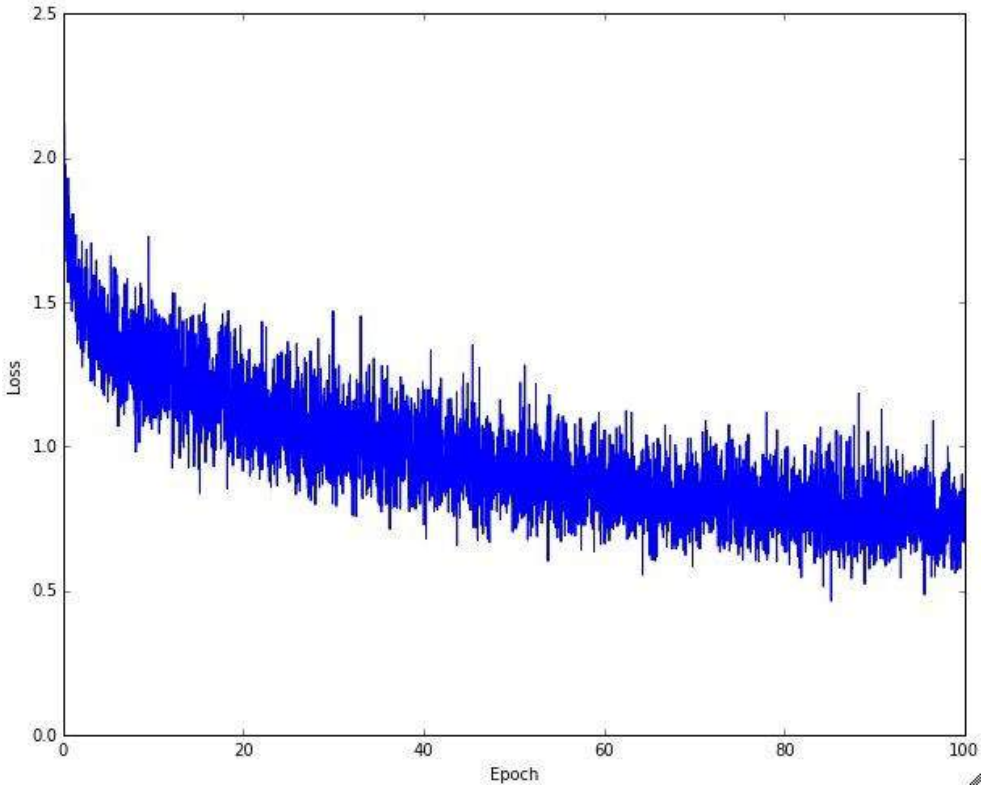
ConvNets: Training

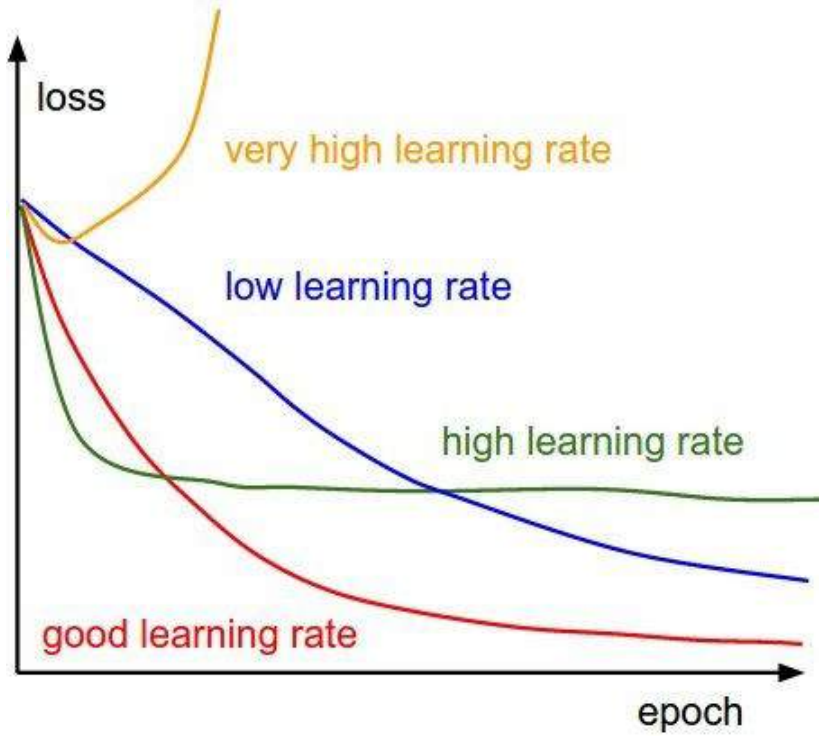


Is this model converging?

Monitor loss during training

ConvNets: Training





Monitor loss during training

Why is training deep neural networks hard?

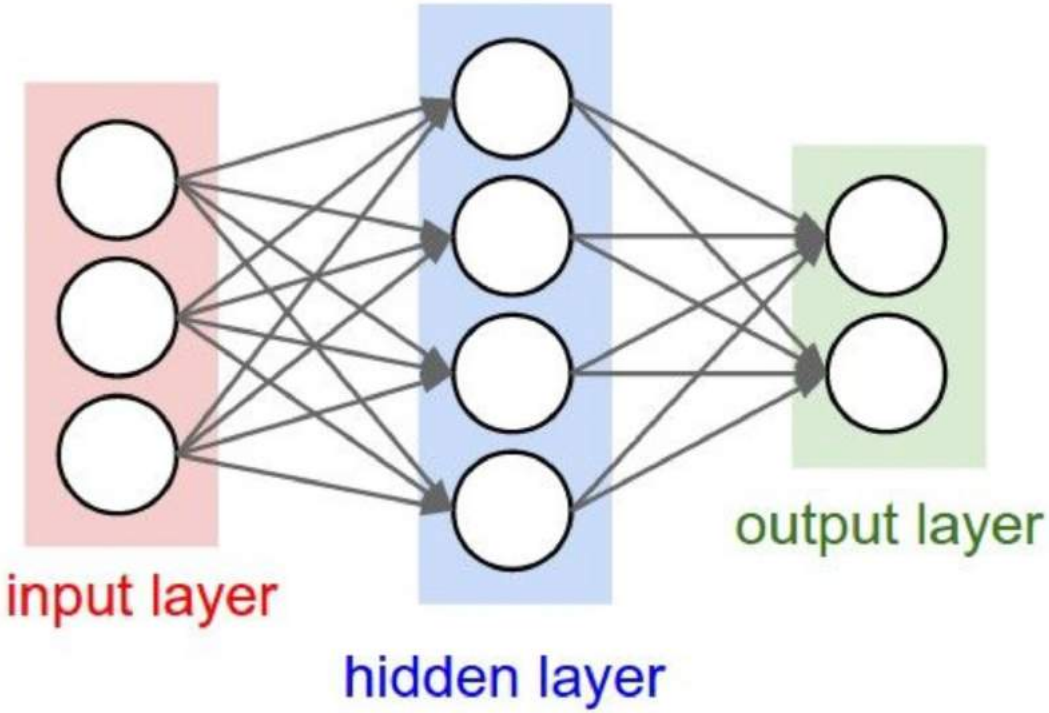
ConvNets: Training

- Loss not decreasing?
- Loss goes to NaN?
- Training diverges?
- Loss is low but testing error is high?
- Trained model outputs constant values?
- My model can’t match the performance in X?
- …

ConvNets Training

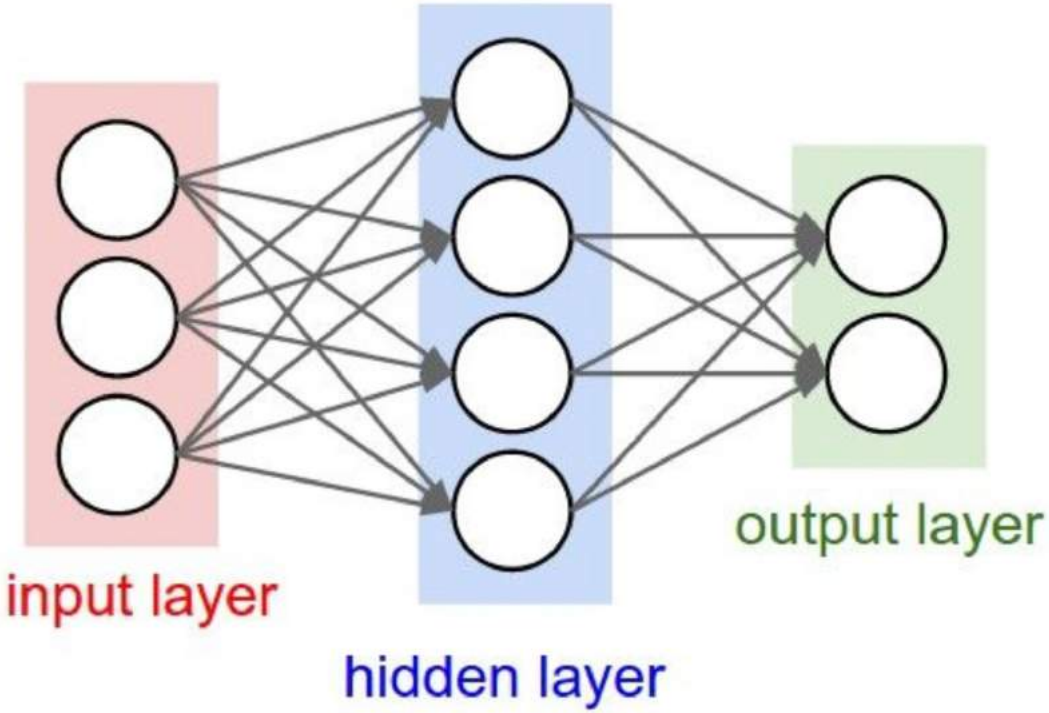
- Initialization
- Regularization
- Optimization

ConvNets Training: Initialization



Some Constant Weight?

ConvNets Training: Initialization



Constant Weight →
Same updates to all units

ConvNets Training: Initialization

random initialization

Small random numbers

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks

ConvNets Training: Initialization

random initialization

Small random numbers

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

for large networks think about
the local gradient (y=wx)

ConvNets Training: Initialization

Xavier initialization

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Var(W)

F-Prop: $Y = WX \rightarrow Var(Y) = D_{in}\, Var(W)\, Var(X)$

If we want $Var(Y) = Var(X)$, select

$$Var(W) = 1 / D_{in}$$

ConvNets Training: Initialization

Xavier initialization

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Var(W)

B-Prop: $dX = W^T dY \rightarrow Var(dX) = D_{out}\, Var(W)\, Var(dY)$

If we want $Var(dY) = Var(dX)$, select

$$Var(W) = 1 / D_{out}$$

ConvNets Training: Initialization

Xavier / He initialization

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Var(W)

Xavier: $Var(W) = 2 / (D_{out} + D_{in})$

[Glorot & Bengio]

ConvNets Training: Initialization

Xavier / He initialization

(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Var(W)

What if a nonlinear activation function $Y = \text{ReLU}(WX)$

He et. al $\quad Var(W) = 2 / D_{in}$

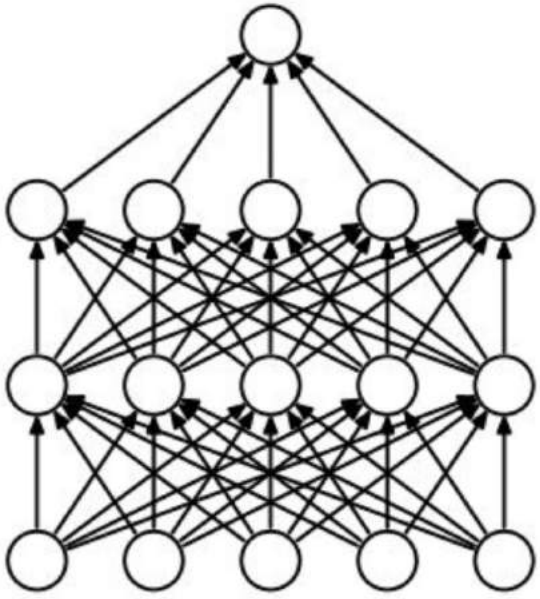
Still an active area of research

[Glorot & Bengio]
[He, Rang, Zhen, Sun]

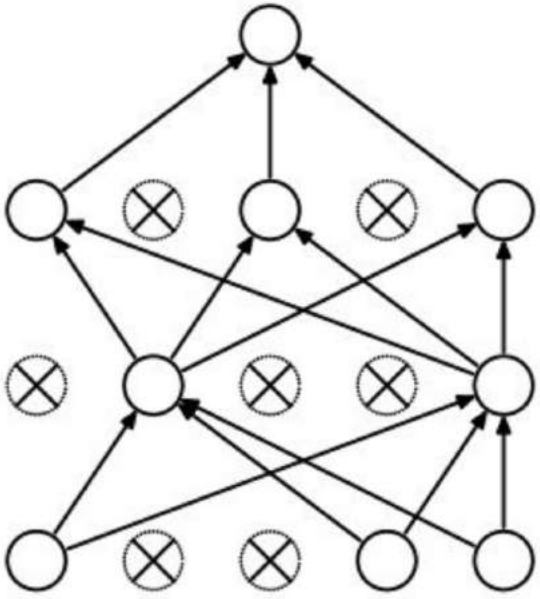
ConvNets Training: Regularization

Regularization: **Dropout**

"randomly set some neurons to zero in the forward pass"



(a) Standard Neural Net



(b) After applying dropout.

[Srivastava et al., 2014]

ConvNets Training: Regularization

"**Batch norm** *is a widely adopted technique that enables faster and more stable training of deep neural networks*"

Consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

[Ioffe and Szegedy, 2015]

Many different normalizations:

- Instance norm
- Layer norm
- Weight norm
- Group norm
- ….

ConvNets Training: Regularization

L2 Regularization

$$\mathcal{L} = \ell + \frac{\lambda}{2} \sum_i ||W_i||_2^2$$

- Adding L2 norm of model parameters to the loss
- Force weights to decay to zero (but not exactly zero)
- $\boldsymbol{\lambda}$ as weight decay (usually a small value, e.g., 1e-4)
- Common technique in learning that helps to regularize the training

ConvNets Training: Regularization

L2 Regularization

$$\mathcal{L} = \ell + \frac{\lambda}{2} \sum_i ||W_i||_2^2$$

Question: What is the gradient for L2 regularization ?

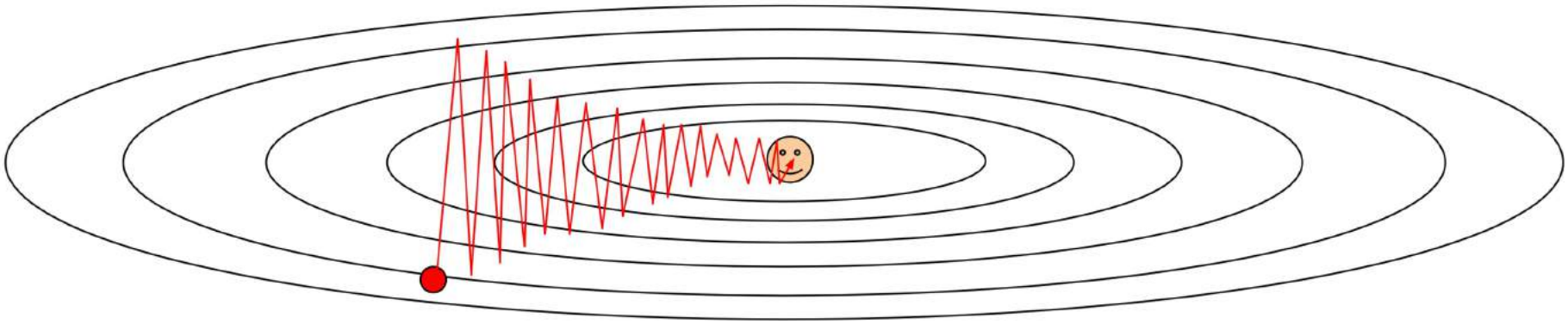
Answer: Itself

Question: How to optimize the network with SGD?

Answer: $$\mathrm{W_i} \leftarrow W_i - \eta \frac{\partial \mathcal{L}}{\partial W_i} - \eta \lambda W_i$$

ConvNets Training: Optimization

Suppose loss function is steep vertically but shallow horizontally:



smoothen gradients using momentum

ConvNets Training: Optimization

Stochastic Gradient Descent (on mini-batches):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial L}{\partial \boldsymbol{\theta}}, \eta \in (0,1)$$

Stochastic Gradient Descent with Momentum:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\Delta}$$

$$\boldsymbol{\Delta} \leftarrow 0.9 \boldsymbol{\Delta} + \frac{\partial L}{\partial \boldsymbol{\theta}}$$

Need to reduce the learning rate once in a while!

Note: there are many other variants...

ConvNets Training: Optimization

- Adagrad [Duchi, Hazan, Singer]
- RMSProp [Hinton's lecture]
- AdaDelta [Zeiler]
- Adam [Kingma, Ba]

Nothing can beat SGD + Momentum with hand-tuned learning rate schedules

ConvNets Training: Optimization

Adam [Kingma, Ba]

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Default parameters: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$

ConvNets Training: Optimization

Two tricks?

- Gradient Clipping
  - Estimate the gradient magnitude for the weights (parameters)
  - Re-normalize the gradients if its magnitude is larger than a pre-specified value (clip the magnitude of the gradient)
  - Gradient direction remain the same
  - Gradient won't explode!

- Warm-up Training
  - Start the training with learning rate = 0
  - Gradually increase the learning rate to the initial learning rate
  - Help the optimizer to escape from a bad initial region

ConvNets Training: Recap

Training diverges:

- Learning rate may be too large → decrease learning rate
- BPROP / Loss is buggy → numerical gradient checking
- Try the tricks (gradient clipping / warmup)

Network is underperforming

- Visualize hidden units/params → fix optimization
- Check your learning rate schedule
- Normalization / Dropout / Data Augmentation / L2 …

Network is too slow

- Compute flops and nr. params. → GPUs

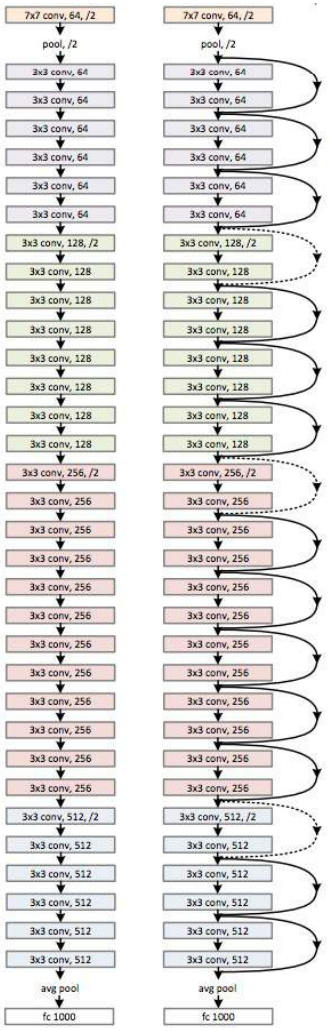
Loss is minimized but accuracy is low

- Check loss function:
  - Is it appropriate for the task you want to solve?
  - Does it have degenerate solutions? Check “pull-up” term

Why is training deep neural networks easy?

Learning ConvNets

- Over-parameterized models
  (#params >> #samples)
- Learned model can memorize training samples!
  [Zhang, Bengio, Hardt, Recht, Vinyals]
- Learned model still be able to generalize to novel samples
- **Why NOT overfitting?**

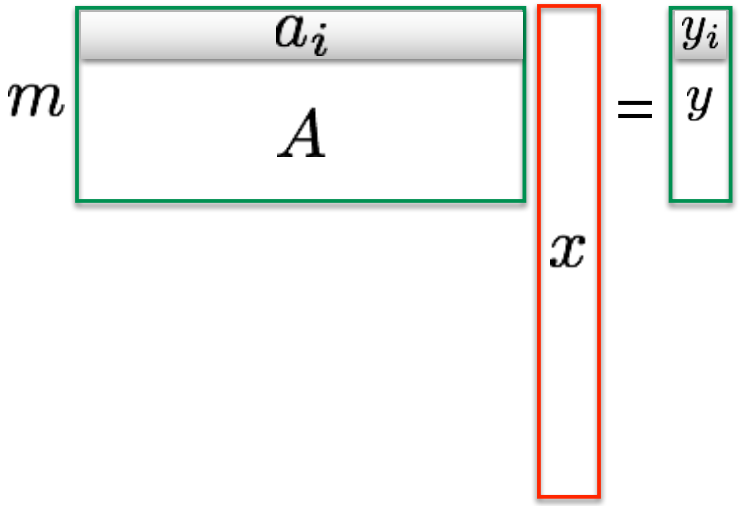


Least squares

$$\min_x f(x) = \|Ax - y\|_2^2$$

Gradient descent initialized at x_0

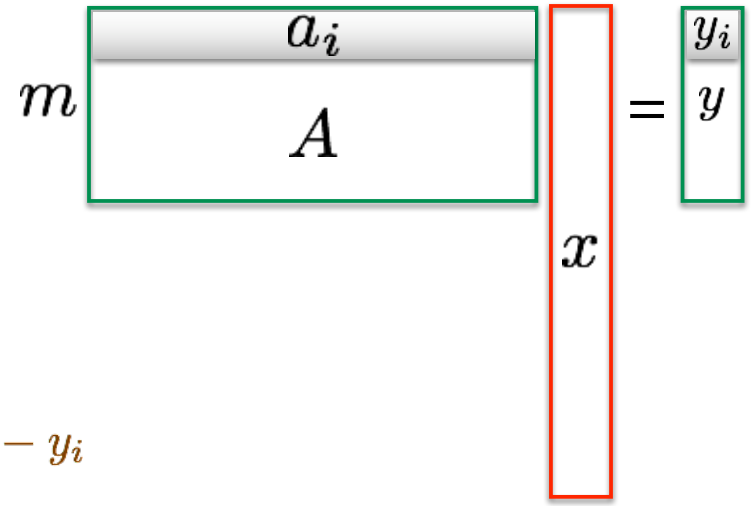
$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$



Slide Credit: Suriya Gunasekar

Least squares

$$\min_x f(x) = \|Ax - y\|_2^2$$



Gradient descent initialized at x_0

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

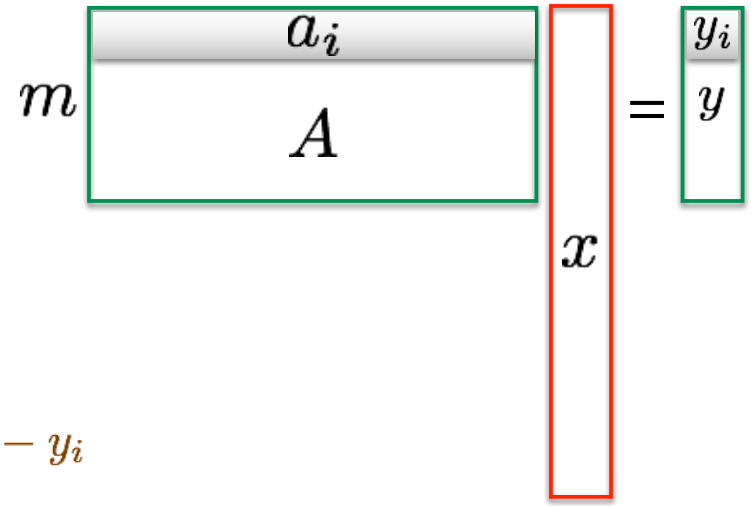
$$\nabla_x f(x) = A^\top (Ax - y) = \sum_{i=1}^{m} r_i a_i, \text{ where } r_i = a_i^\top x - y_i$$

Gradient based updates will only change component of x_0 along the span of a_i's

Slide Credit: Suriya Gunasekar

Least squares

$$\min_x f(x) = \|Ax - y\|_2^2$$



Gradient descent initialized at x_0

$$x_{t+1} = x_t - \eta \nabla_x f(x_t)$$

$$\nabla_x f(x) = A^\top (Ax - y) = \sum_{i=1}^{m} r_i a_i, \text{ where } r_i = a_i^\top x - y_i$$

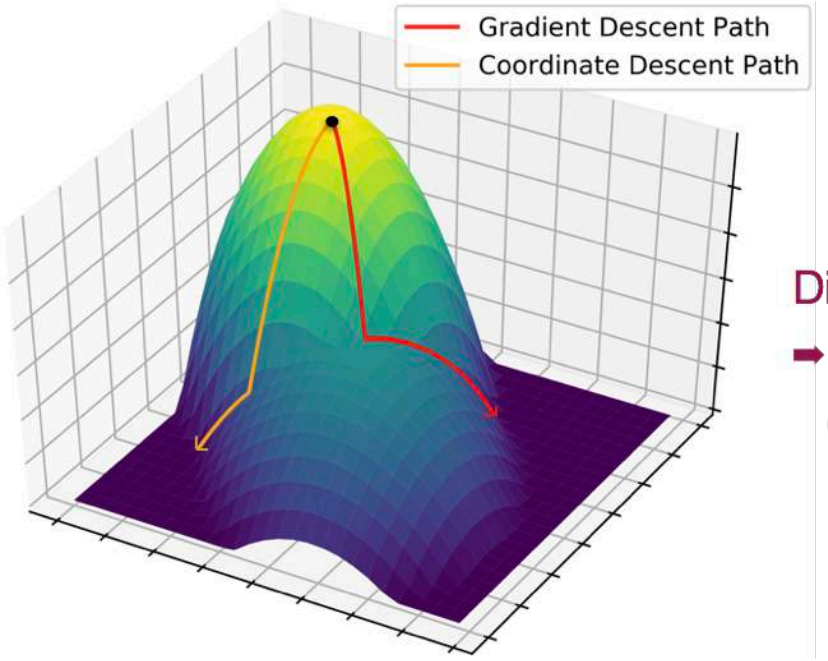
Gradient based updates will only change component of x_0 along the span of a_i's

$$\text{If} \quad x_0 = 0, \quad \text{then} \quad \widehat{x}_{(s)gd} = \underset{Ax=y}{\operatorname{argmin}} \|x\|_2$$

Slide Credit: Suriya Gunasekar

Implicit Regularization

Inductive bias induced by choice of optimization algorithm



Different global optima
➡ different generalization (test error)

Choice of optimization algorithm ⟺ **Implicit bias towards "good" global optima**

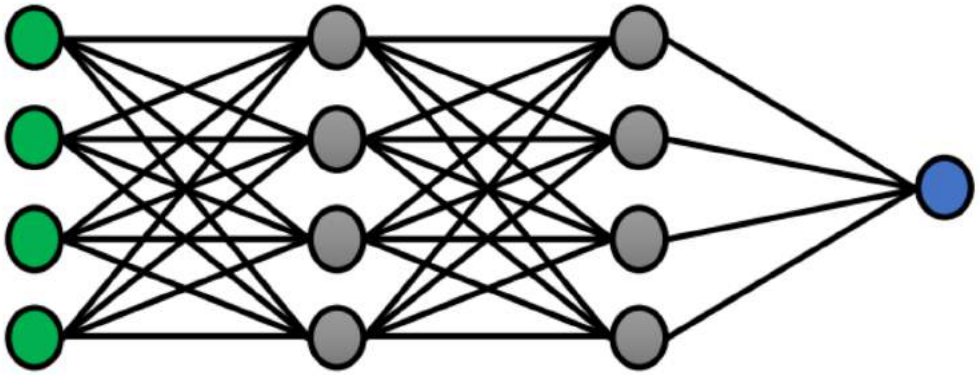
Slide Credit: Suriya Gunasekar

Goal

For DNNs, we want to characterize the "complexity" implicitly minimized by common optimization algorithms?

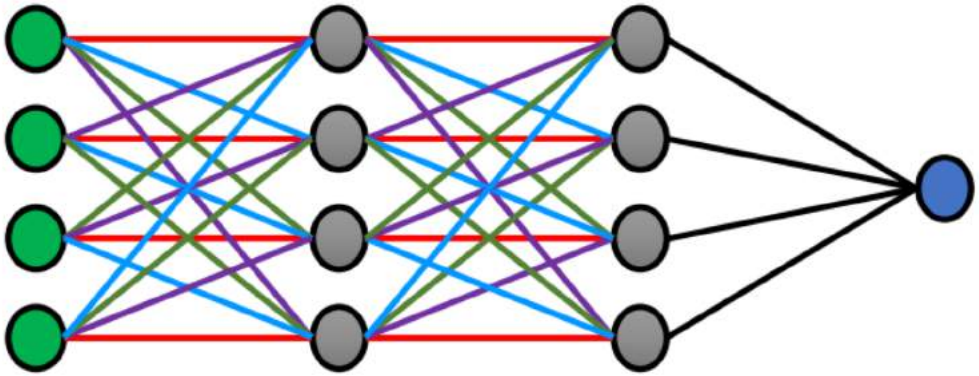
Slide Credit: Suriya Gunasekar

Goal



(a) Fully connected network of depth L

$\overline{\boldsymbol{\beta}}^{\infty} \propto \underset{\forall n,\, y_n \langle \mathbf{x}_n, \boldsymbol{\beta} \rangle \geq 1}{\operatorname{argmin}} \|\boldsymbol{\beta}\|_2$ (independent of L)



(b) Convolutional network of depth L

$\overline{\boldsymbol{\beta}}^{\infty} \propto$ first order stationary point of $\underset{\forall n,\, y_n \langle \mathbf{x}_n, \boldsymbol{\beta} \rangle \geq 1}{\operatorname{argmin}} \|\widehat{\boldsymbol{\beta}}\|_{2/L}$

Linear models:

- Converge to large margin solutions

Linear ConvNets:

- Learning frequency sensitive features

ConvNets:

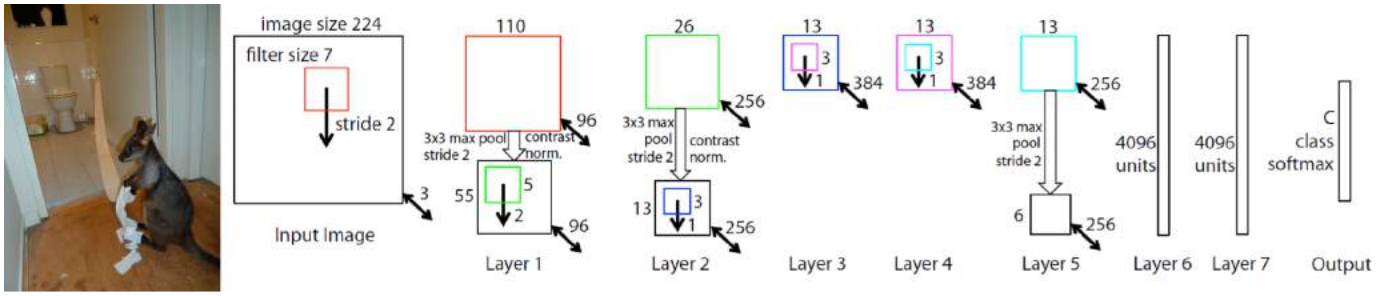
- ??
- [Allen-Zhu, Li, Liang]

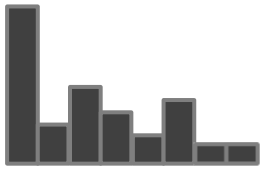
[Gunasekar, Lee, Soudry, Srebro]

Visualizing and Understanding Convolutional Networks

Many figures from Matt Zeiler

Visualizing ConvNets





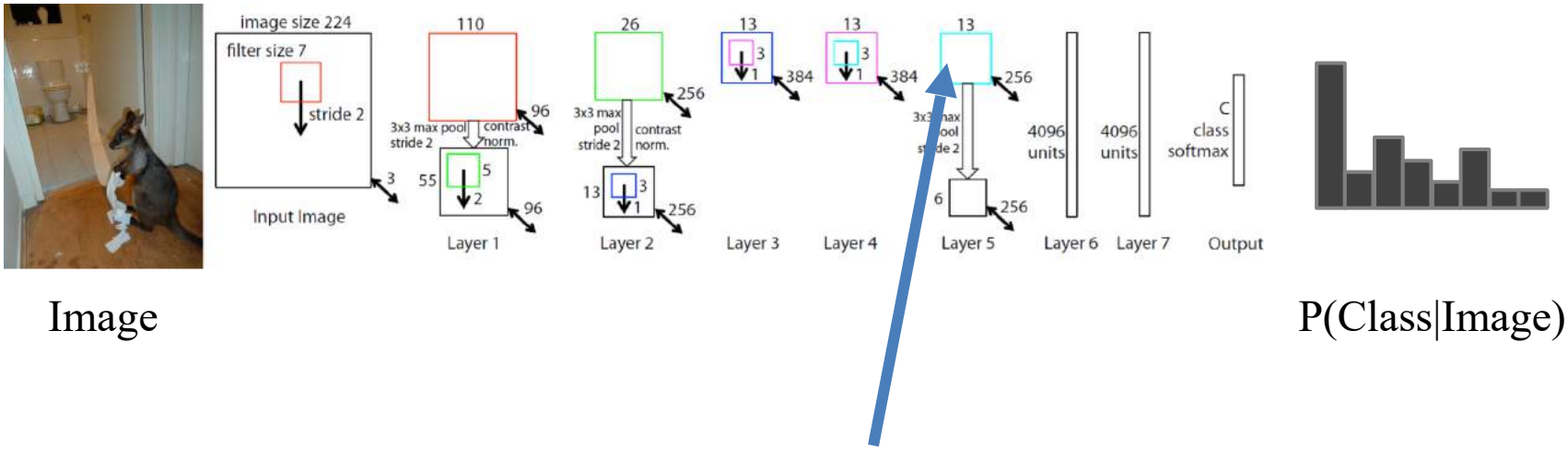
Image

P(Class|Image)

Visualizing ConvNets

- “It’s a black box!”
- “Nobody understands what’s going on!”
- “Conv1 is gabor filters, but what’s actually going on?!”
- “Sure, LeCun and Hinton know how to make them work, but it’s magic.”

Visualizing ConvNets: Goal

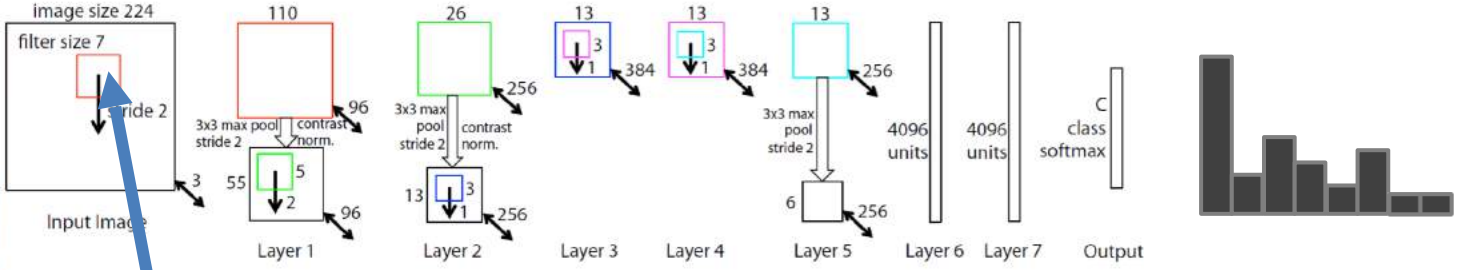


Image

P(Class|Image)

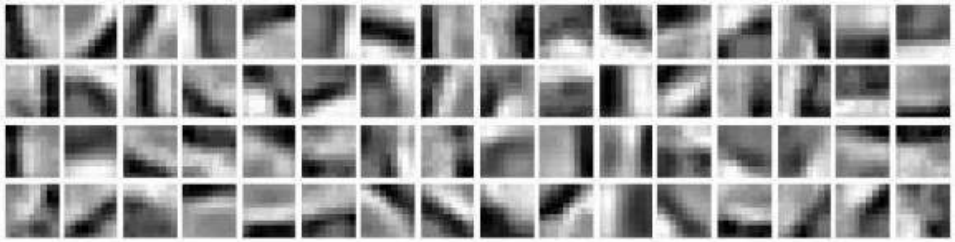
What does this neuron mean?

Visualizing ConvNets: One Solution



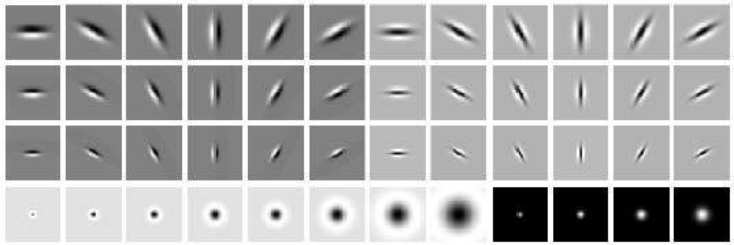
Image

P(Class|Image)



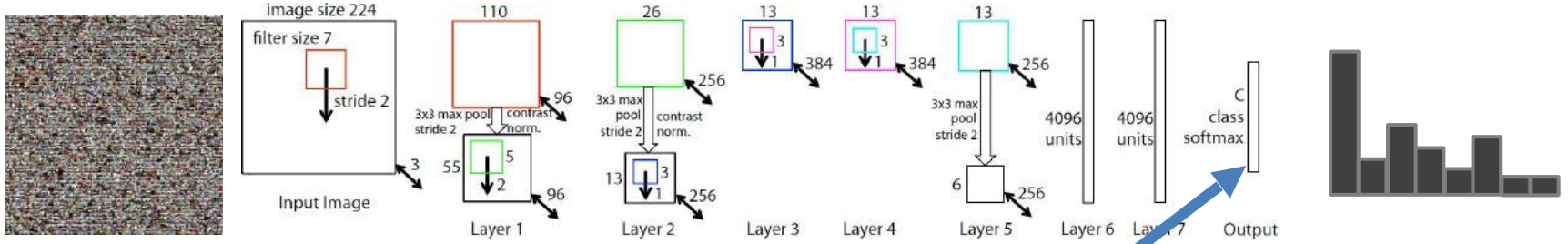
Ranzato et al. ‘07

Compare With:



Leung and Malik ‘01

Visualizing ConvNets: A Simple Scheme



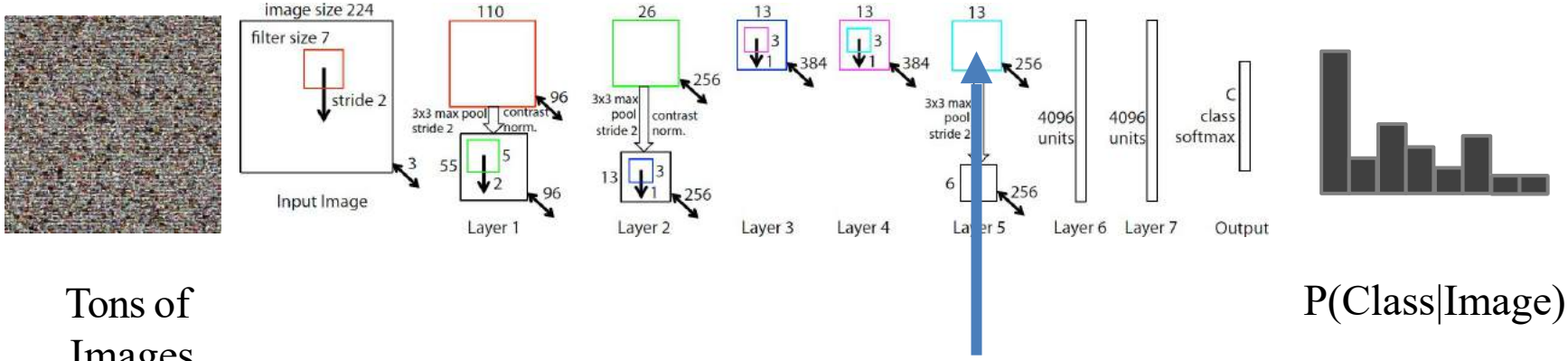
Tons of Images

P(Class|Image)

Most Wallaby-like

Least Wallaby-like

Visualizing ConvNets: A Simple Scheme

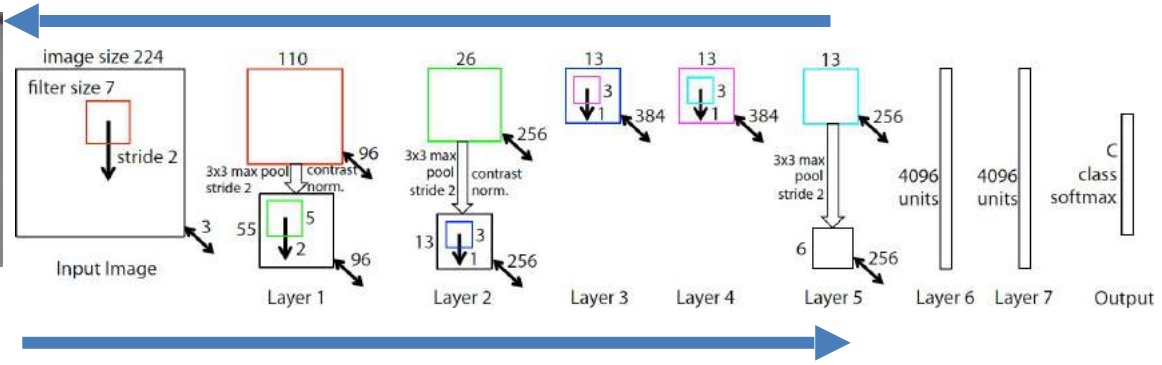


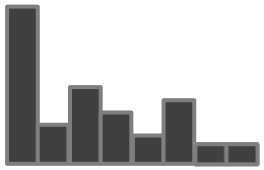
Tons of Images

P(Class|Image)

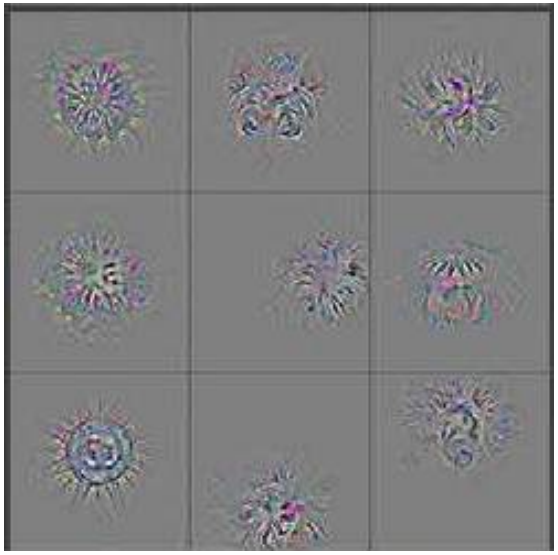
Visualizing ConvNets: What is really going on?

Max-Response Image

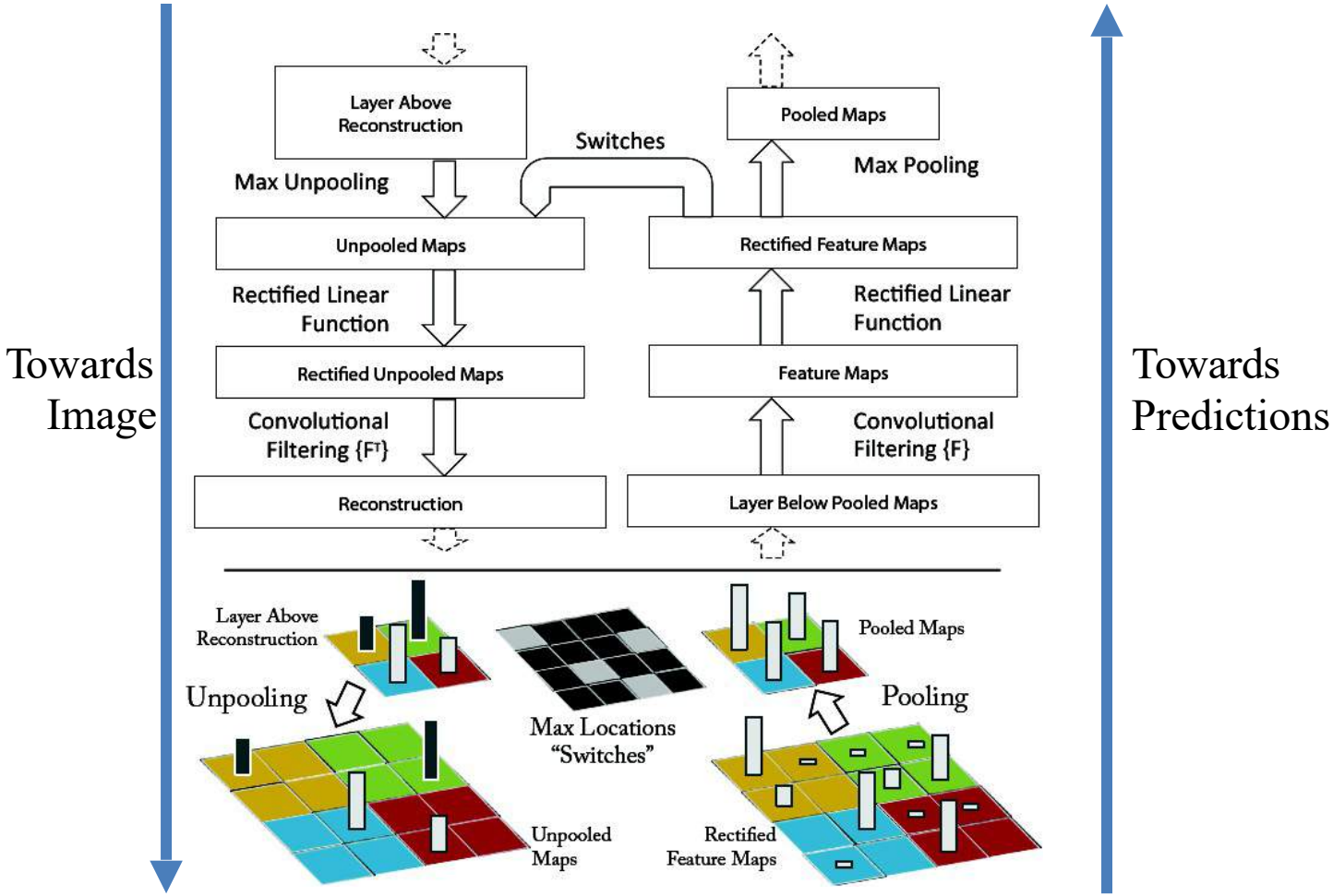




P(Class|Image)



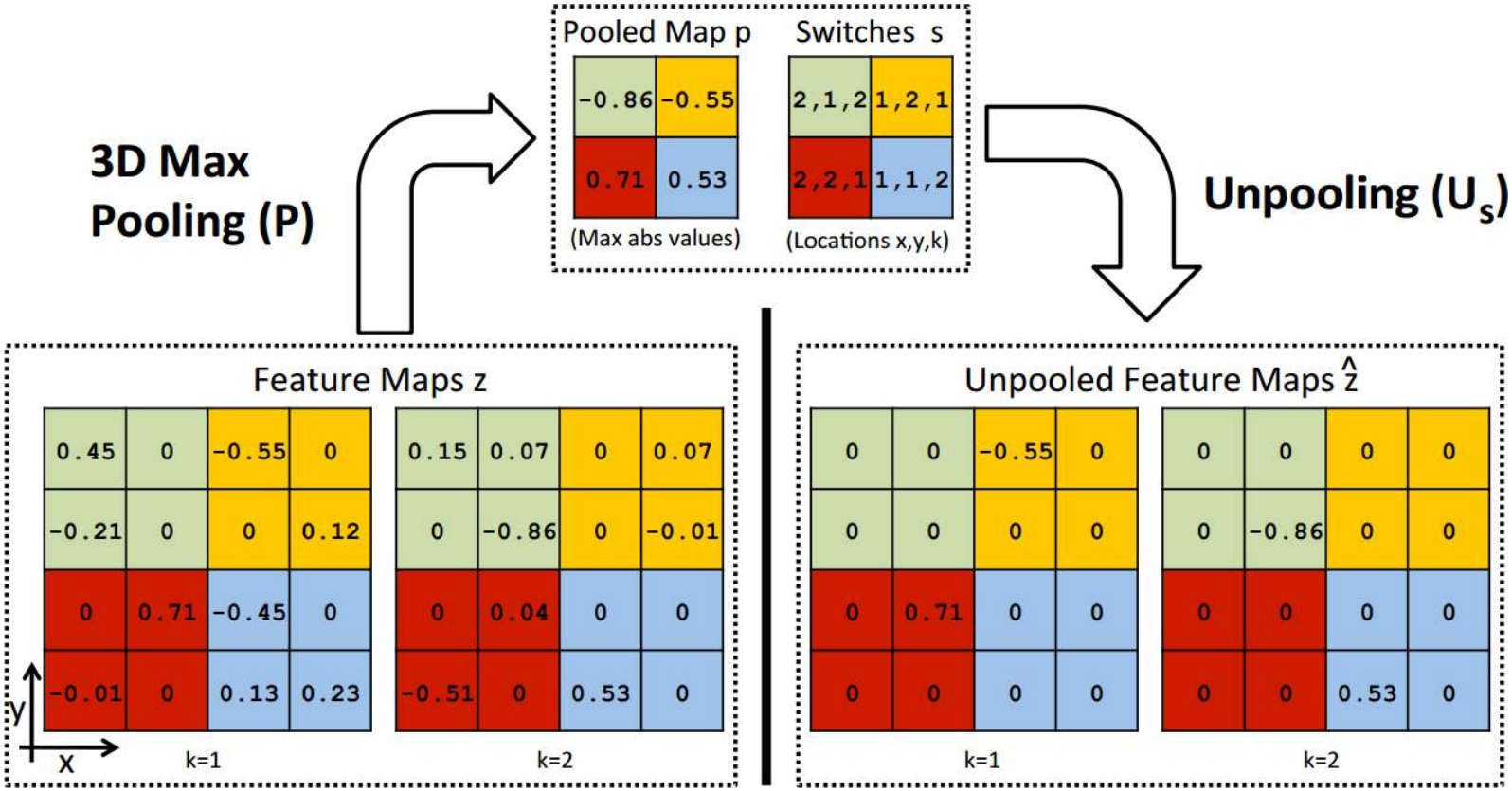
Visualizing ConvNets: Going back to images



Visualizing ConvNets: Things to invert

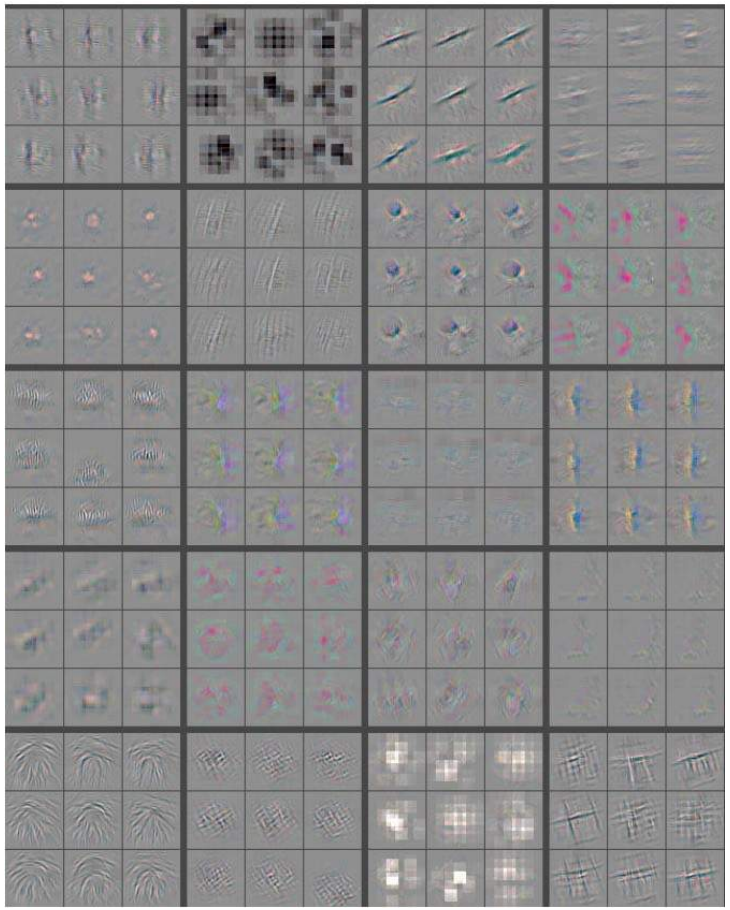
- Convolutions/Filtering
- Rectification/Non-linearity
- Pooling

Visualizing ConvNets: Things to invert

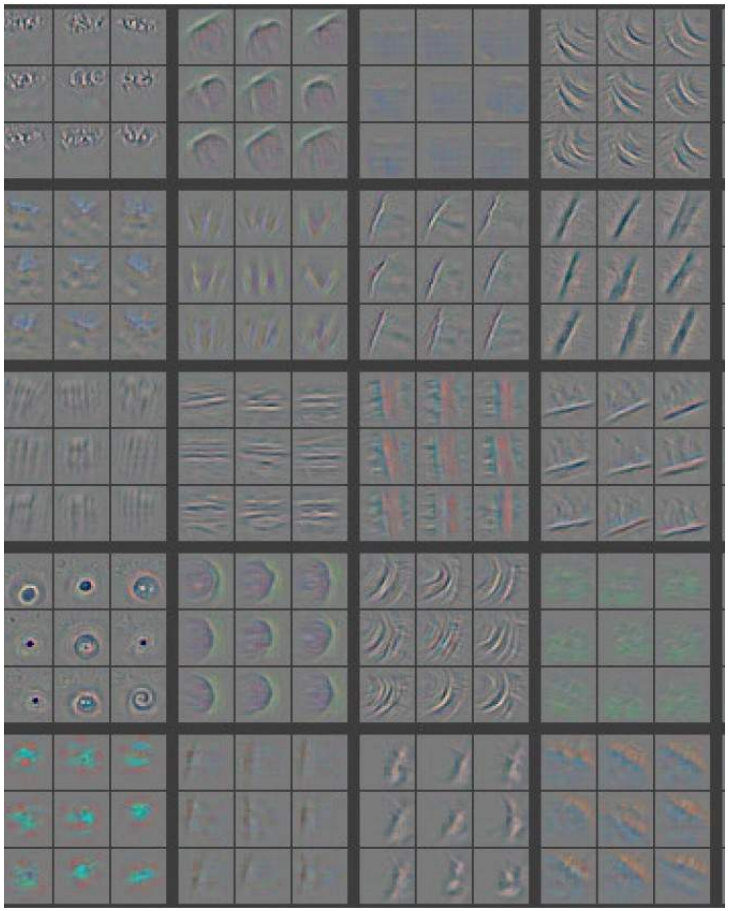


Visualizing ConvNets:

A tour through the network



Alexnet



Zeiler and Fergus
+1.7% Accuracy