

Deep Learning

A course about theory & practice

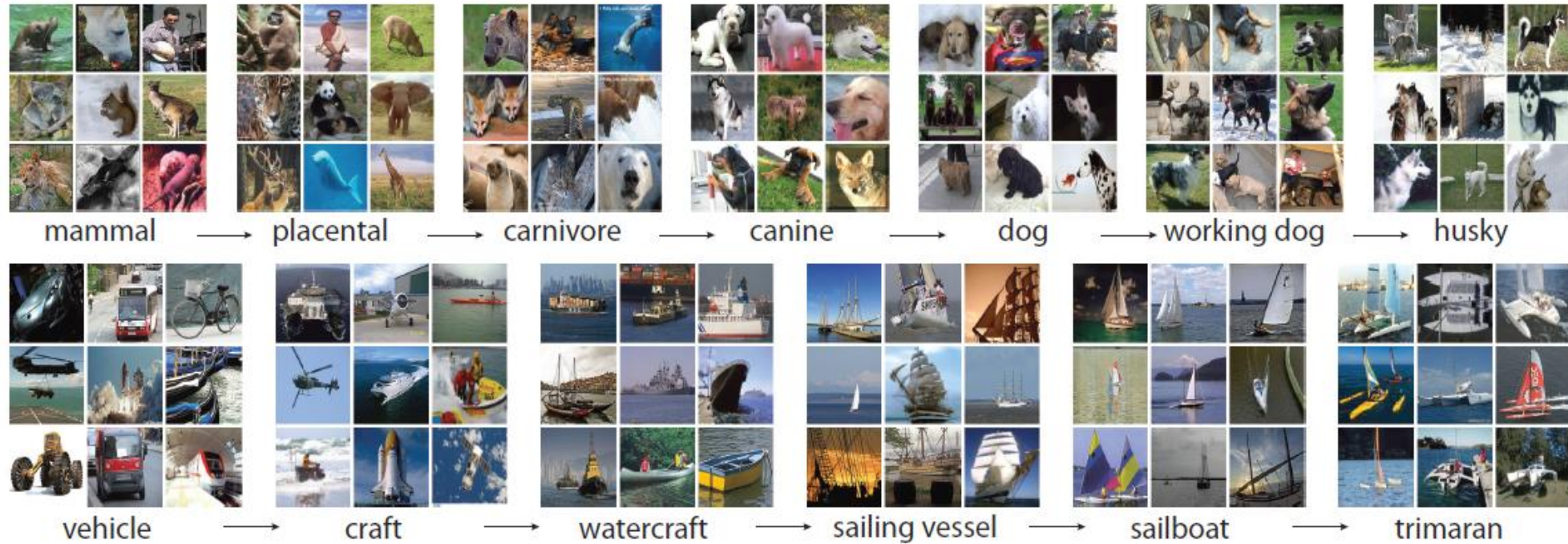


Deep Convolutional Neural Networks

Marco Piastra

ImageNet Challenge

- The ImageNet Large Scale Visual Recognition Challenge



1,461,406 full resolution images

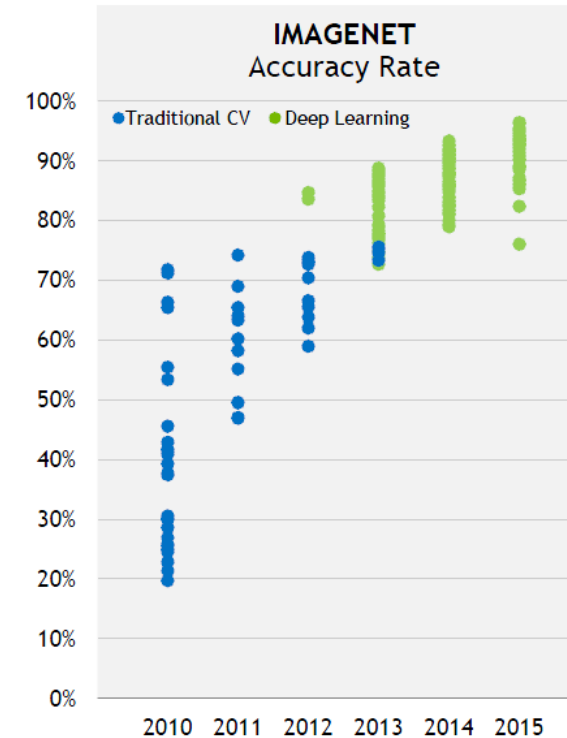
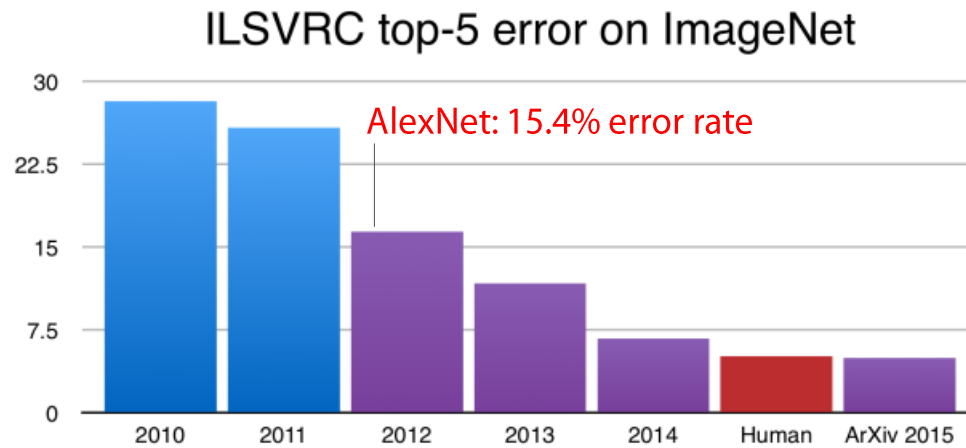
Complex and multiple textual annotation,
hierarchy of 1000 object classes along several dimensions

The image classification challenge is run annually since 2010

[figures from www.nvidia.com]

ImageNet Challenge

- The ImageNet Large Scale Visual Recognition Challenge



1,461,406 full resolution images

Complex and multiple textual annotation,
hierarchy of 1000 object classes along several dimensions

The image classification challenge is run annually since 2010

[figures from www.nvidia.com]

The Mother of all DCNNs

Deep Convolutional Neural Network (DCNN)

- **AlexNet** [Krizhevsky, Sutskever & Hinton, 2012]

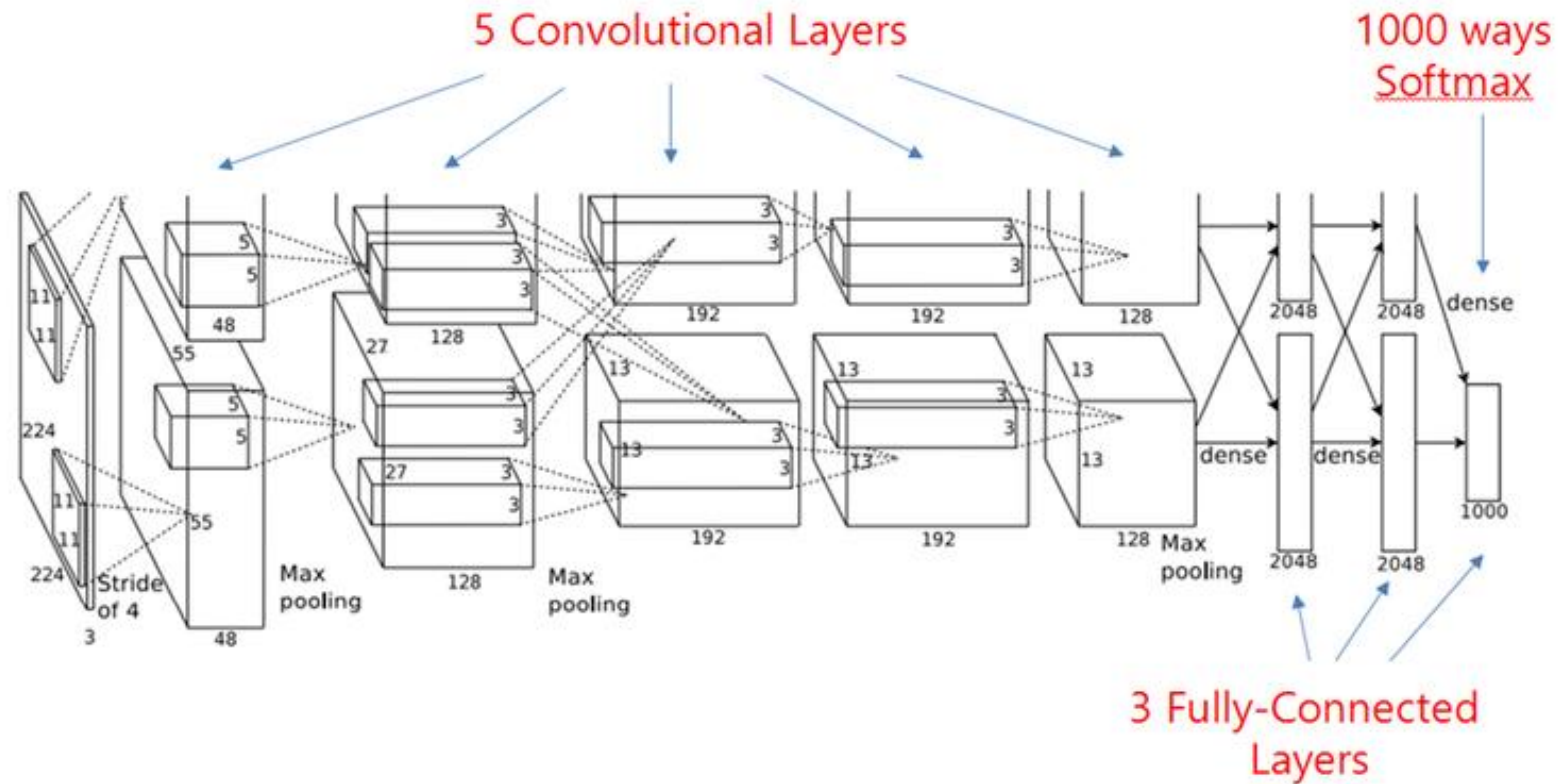
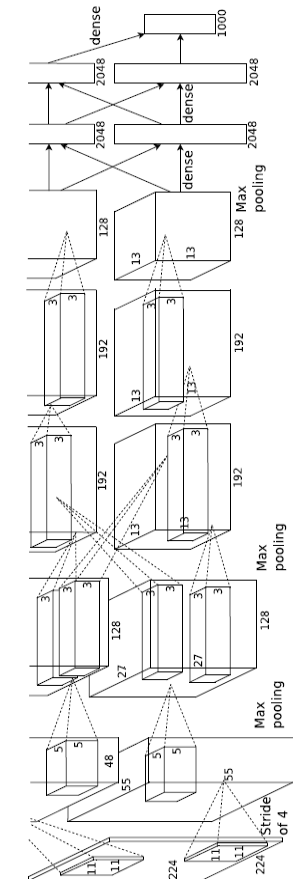
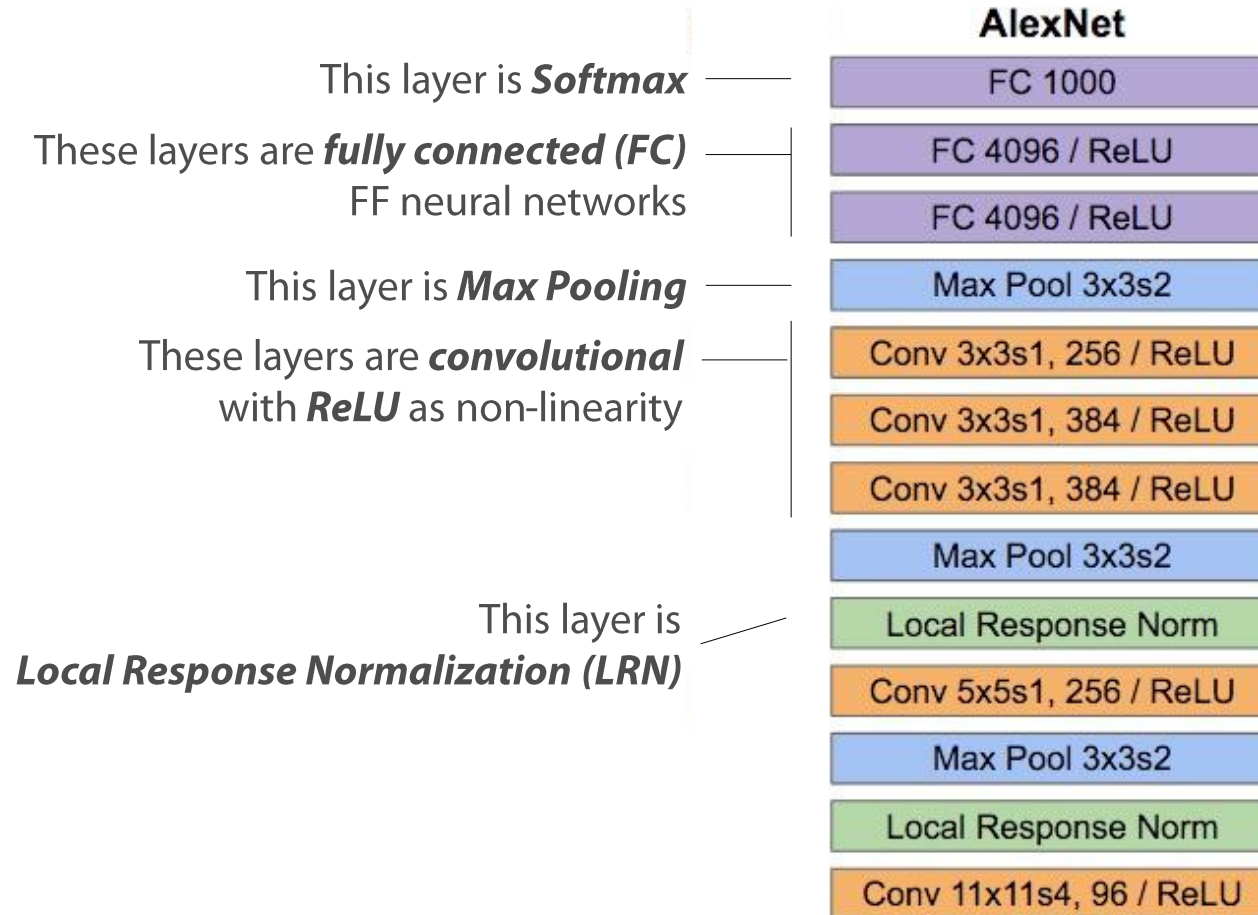


Image from [Krizhevsky, Sutskever & Hinton, 2012]

The Mother of all DCNNs

Deep Convolutional Neural Network (DCNN)

- **AlexNet** [Krizhevsky, Sutskever & Hinton, 2012]



DCNN Building Blocks (layerwise)

Convolutional Layer

■ Convolution operation

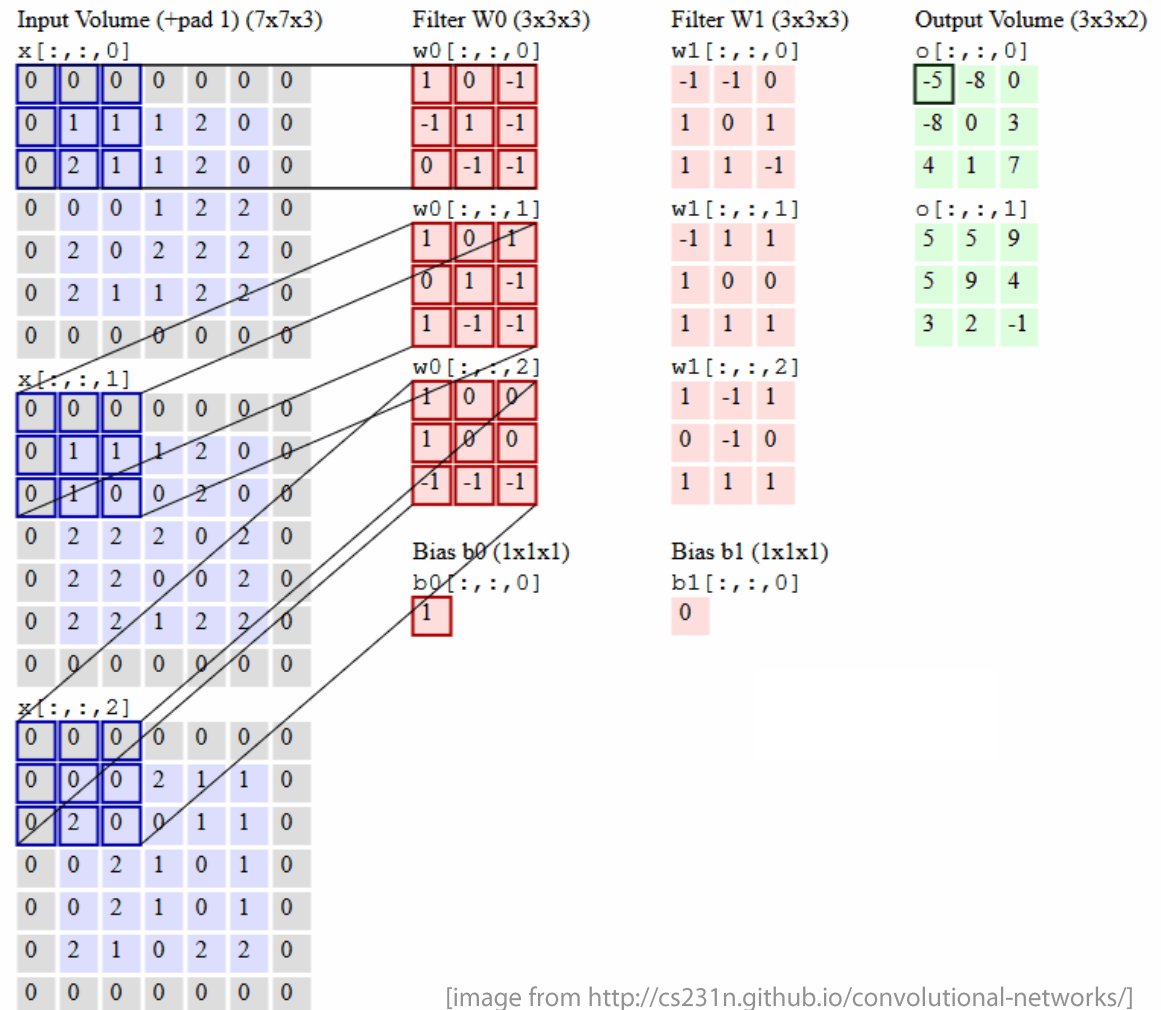
A *convolution filter*

is a square (or cubic) matrix

It is first centered on a pixel of the input image

It produces a scalar value: the dot product between the filter and the image region around the pixel

By mapping the same procedure on all pixels of the input image, a new image is produced (i.e. a *feature map*)



[image from <http://cs231n.github.io/convolutional-networks/>]

Convolutional Layer

Convolution operations (on images)

A **convolution filter**

is a square (or cubic) matrix

In symbols

$$Y_i = W_i * X$$

Y_i : i -th feature map

$*$: convolution operator

where:

X : Input image (e.g. RGB)

Input Volume (+pad 1) (7x7x3)

$x[:, :, 0]$

0	0	0	0	0	0	0
0	2	1	2	1	0	0
0	1	2	1	2	0	0
0	0	2	1	2	2	0
0	2	0	2	0	0	0
0	1	1	0	1	1	0
0	0	0	0	0	0	0

$x[:, :, 1]$

0	0	0	0	0	0	0
0	1	1	1	2	0	0
0	1	1	1	0	2	0
0	0	2	0	1	0	0
0	2	0	0	0	1	0
0	2	1	1	1	1	0
0	0	0	0	0	0	0

$x[:, :, 2]$

0	0	0	0	0	0	0
0	2	0	1	1	1	0
0	1	0	0	0	1	0
0	1	0	0	1	1	0
0	0	0	1	2	0	0
0	0	0	2	0	2	0
0	0	0	0	0	0	0

Filter W0 (3x3x3)

$w0[:, :, 0]$

0	0	-1
0	0	0
1	-1	1

$w0[:, :, 1]$

0	0	-1
-1	-1	-1
-1	-1	0

$w0[:, :, 2]$

1	1	1
0	-1	-1
0	-1	-1

Bias b0 (1x1x1)

$b0[:, :, 0]$

1

W_0

Filter W1 (3x3x3)

$w1[:, :, 0]$

-1	0	0
0	0	1
1	1	1

$w1[:, :, 1]$

-1	0	0
0	-1	-1
1	0	1

$w1[:, :, 2]$

-1	-1	-1
-1	0	0
1	0	1

Bias b1 (1x1x1)

$b1[:, :, 0]$

0

W_1

Convolution filters

Output Volume (3x3x2)

$o[:, :, 0]$

-4	-4	-3
-8	-10	-1
-2	-1	-1

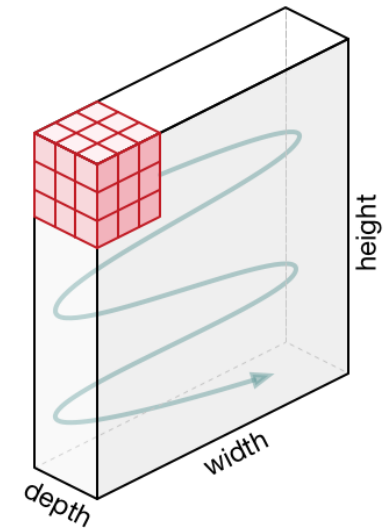
$o[:, :, 1]$

3	4	1
1	2	-2
-2	-4	-3

Y_0

Y_1

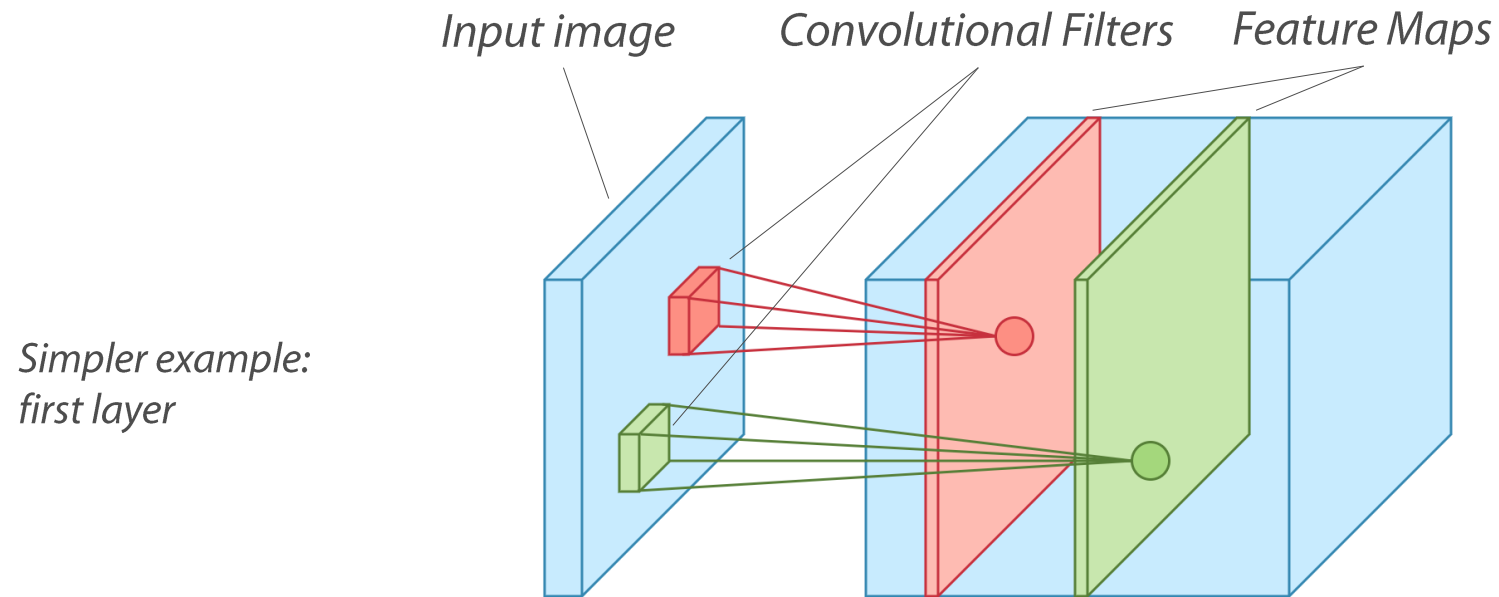
Feature Maps



[image from <http://cs231n.github.io/convolutional-networks/>]

Convolutional Layer

- A bank of convolutional filters



$$\mathbf{X} * [\mathbf{W}_0, \dots, \mathbf{W}_h] = [\mathbf{Y}_0, \dots, \mathbf{Y}_h]$$

[image adapted from <http://cs231n.github.io/convolutional-networks/>]

Convolutional Layer

■ Convolution operation with non-linearity

The linear form for convolution

$$\mathbf{Y}_i := \mathbf{W}_i * \mathbf{X}$$

A non-linearity is added in convolutional neural networks

$$\mathbf{Y}_i := g(\mathbf{W}_i * \mathbf{X})$$

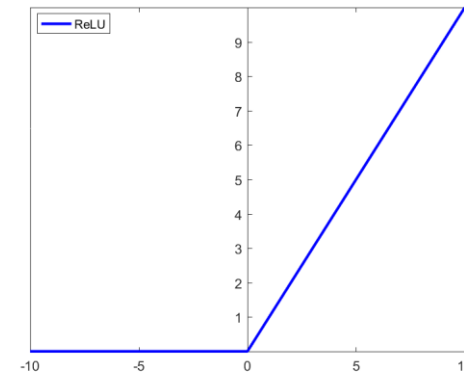
Applied elementwise to all matrix components

$$\mathbf{Y}_i := \text{ReLU}(\mathbf{W}_i * \mathbf{X})$$

Why ReLU?

To be seen later on, when discussing DCNN training

$$y = \max(0, x)$$

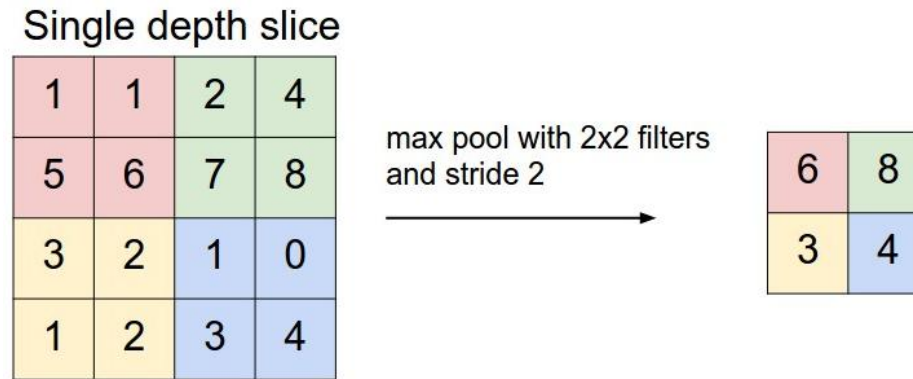


ReLU

Max Pooling Layer

- **Max Pooling operation**

Returns the maximum value in a pre-defined region of its input



Local Response Normalization Layer

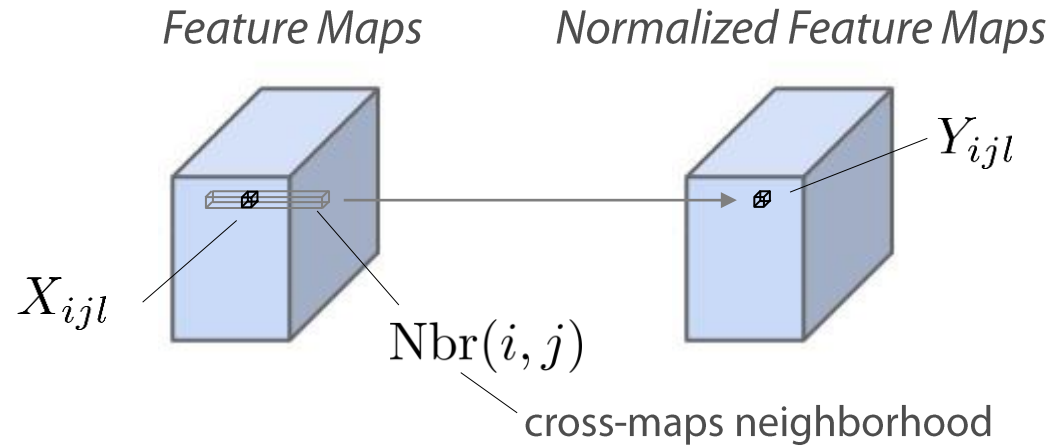
Local Response Normalization (LRN)

Rationale:

compensating the tendency of ReLU to produce large values in output

Two variants:

- *across feature maps* (i.e. as in figure)
- *within feature map* (i.e. with neighboring pixels)



$[X_0, \dots, X_h]$

$[Y_0, \dots, Y_h]$

$$Y_{ijl} := \frac{X_{ijl}}{\left(a + \alpha \sum_{k \in \text{Nbr}(i,j)} (X_{ijk})^2 \right)^\beta}$$

note the summation

where a, α, β are fixed hyperparameters

AlexNet Architecture

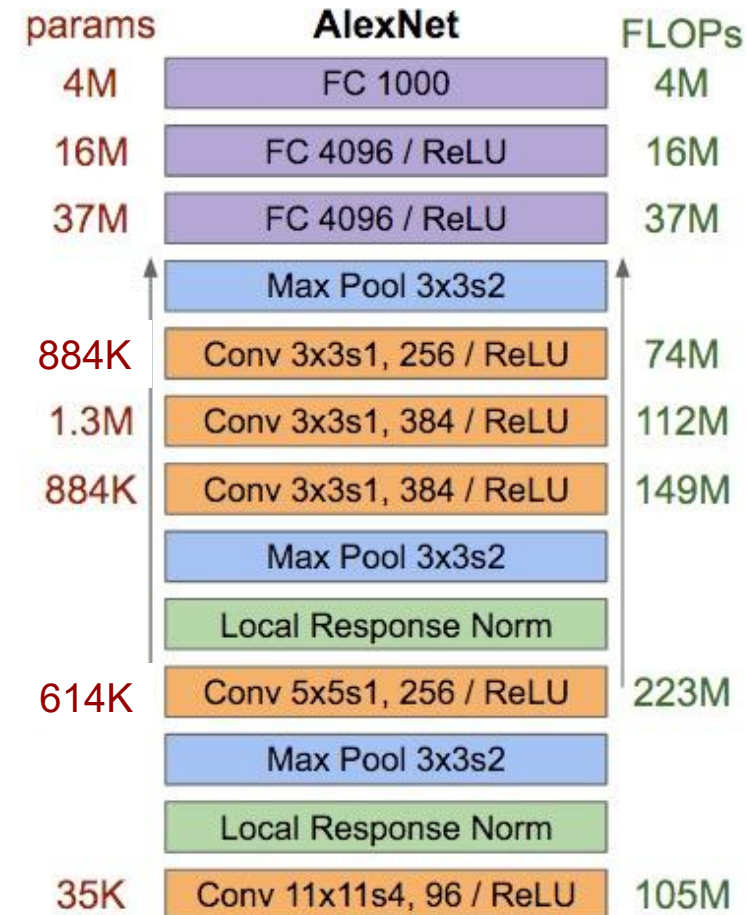
- **AlexNet** [Krizhevsky, Sutskever & Hinton, 2012]

- number of parameters, per layer
in red on the left
- number of *floating point operations*,
(FLOP) per layer in single forward pass
in green on the right

*Higher layers have more parameters
but the bulk of the computation
takes place at lower layers*

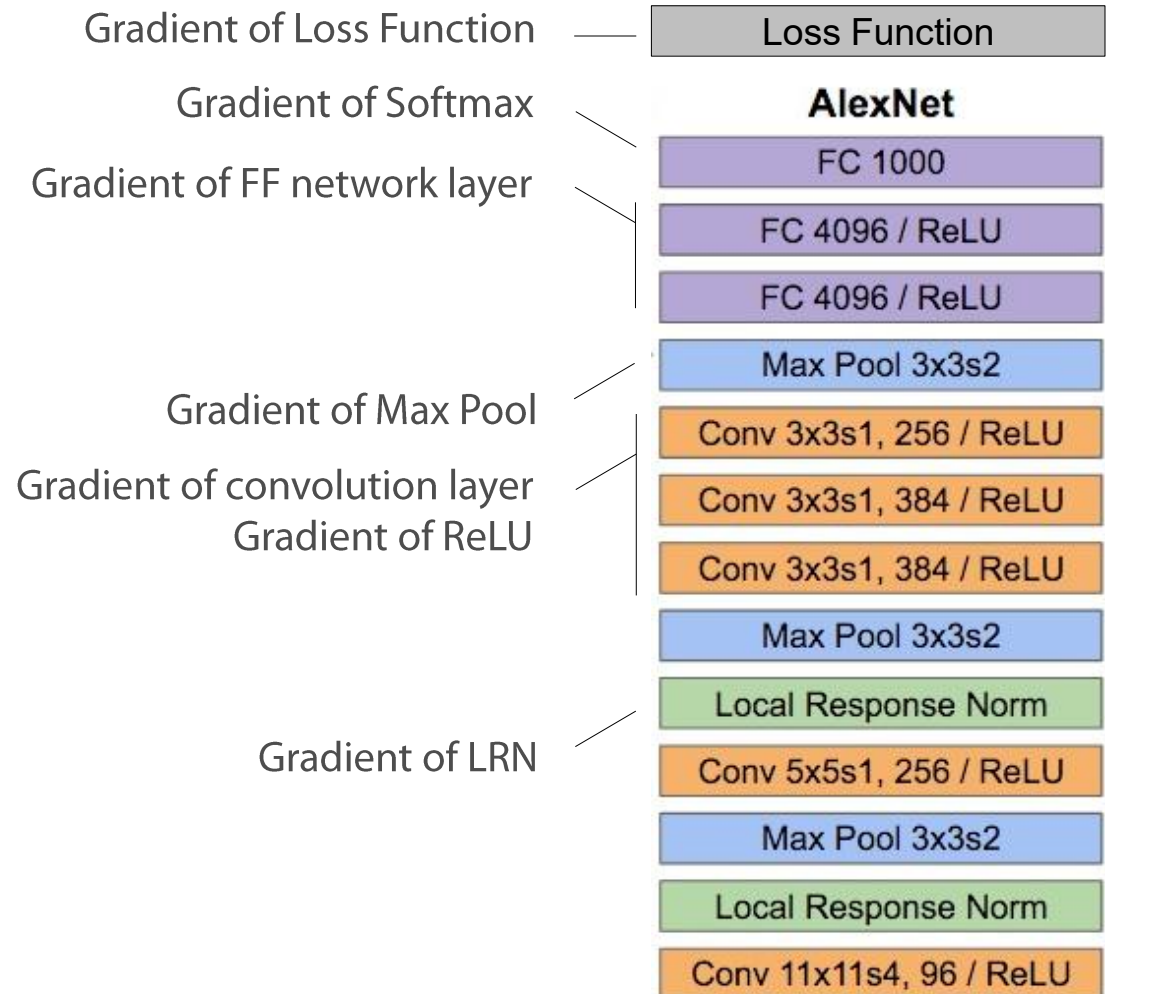
Totals:

- **around 64M parameters**
- **around 837M FLOPs for a single pass**



AlexNet Gradient

■ Computing gradients (backward propagation)



Convolutional Layer Gradient

■ Gradient of convolutional layer

Define

$$Y = (W * X)$$

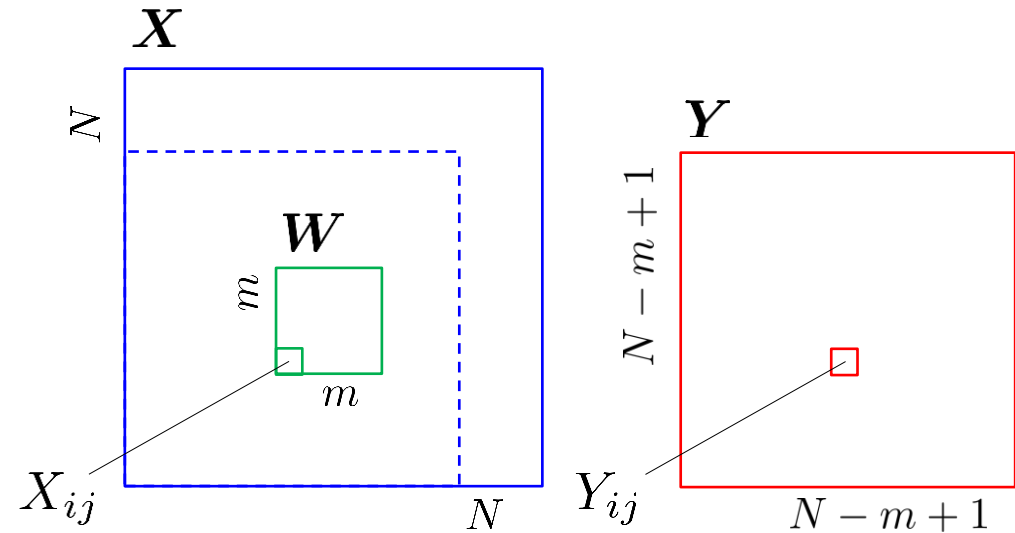
and, for convenience, assume that

$$W \in \mathbb{R}^{m \times m}, \quad X \in \mathbb{R}^{N \times N}$$

the input image is square

$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

*the convolution operator is 'centered'
in the lower left corner*



*All matrices in this example
are indexed as images:
i.e. the lower left corner is 0^o0*

(In general m is odd and the convolution is 'centered' in the centroid of the filter)

Convolutional Layer Gradient

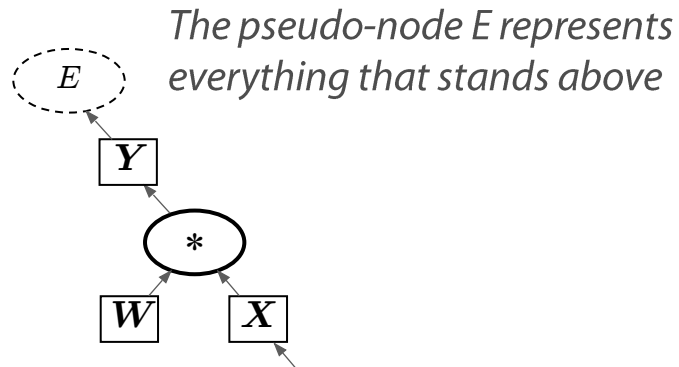
■ Gradient of convolutional layer

$$Y = (W * X)$$

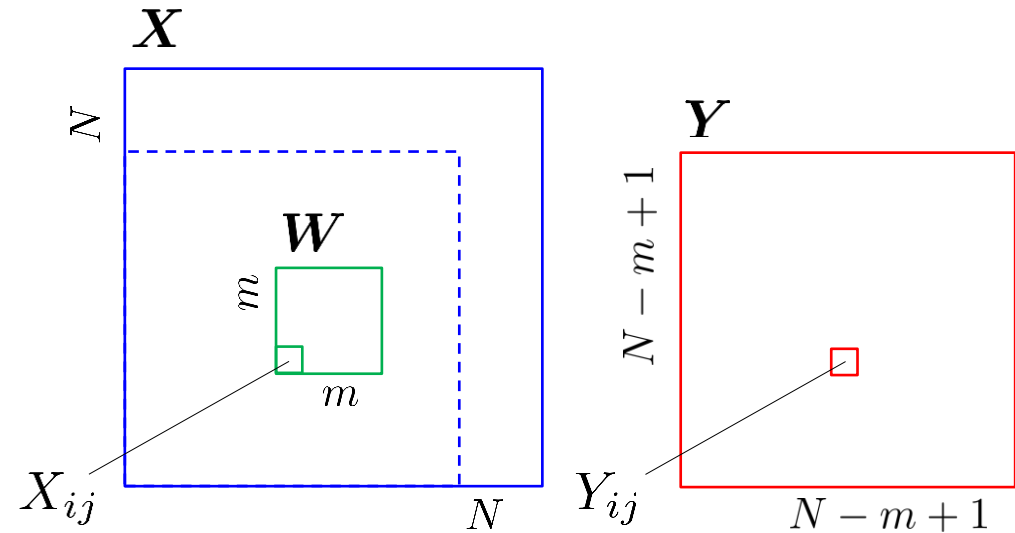
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Consider the pseudo-graph

The flow is bottom-up, in this example



Another subgraph stands below (not represented)



Convolutional Layer Gradient

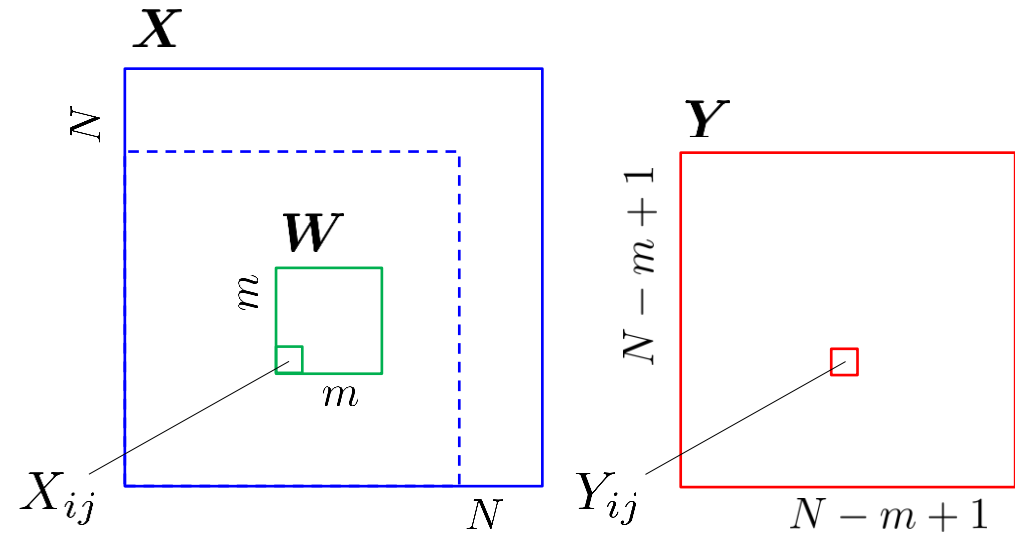
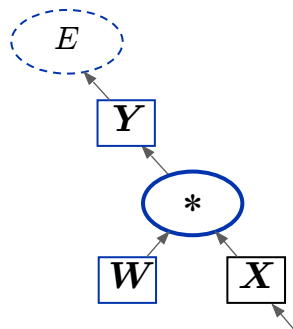
■ Gradient of convolutional layer

$$Y = (W * X)$$

$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Case 1:

$$\frac{\partial}{\partial W} E(Y) \quad \text{i.e. the 'end of the chain'}$$



Convolutional Layer Gradient

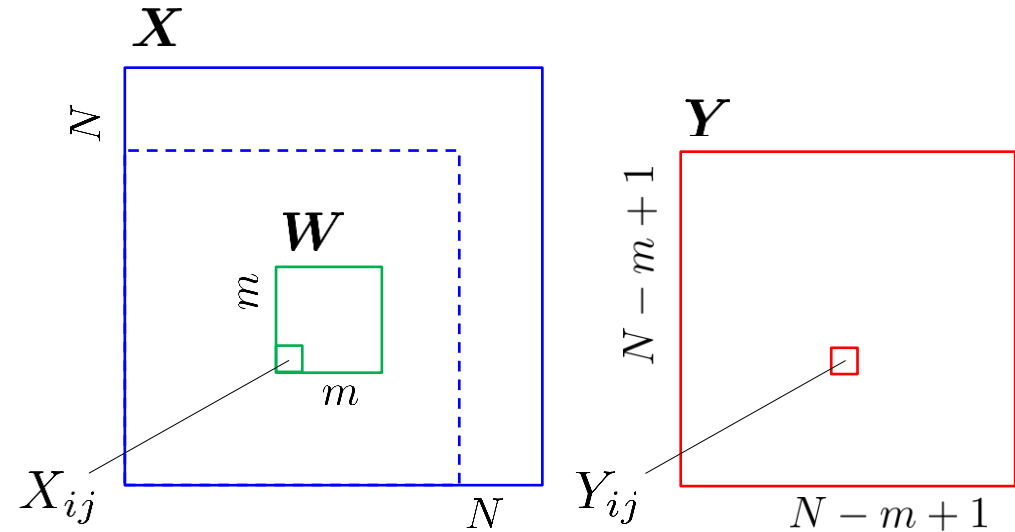
■ Gradient of convolutional layer

$$Y = (W * X)$$

$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

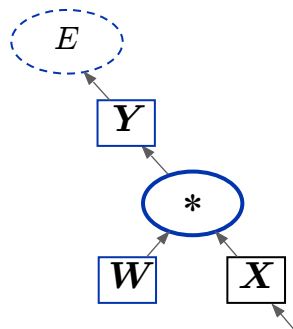
Case 1:

$$\frac{\partial}{\partial W} E(Y) \quad \text{i.e. the 'end of the chain'}$$



$$\frac{\partial}{\partial W_{lk}} E(Y) = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E(Y)}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial W_{lk}}$$

by applying the chain rule
in extended version (see also Episode 1)



Convolutional Layer Gradient

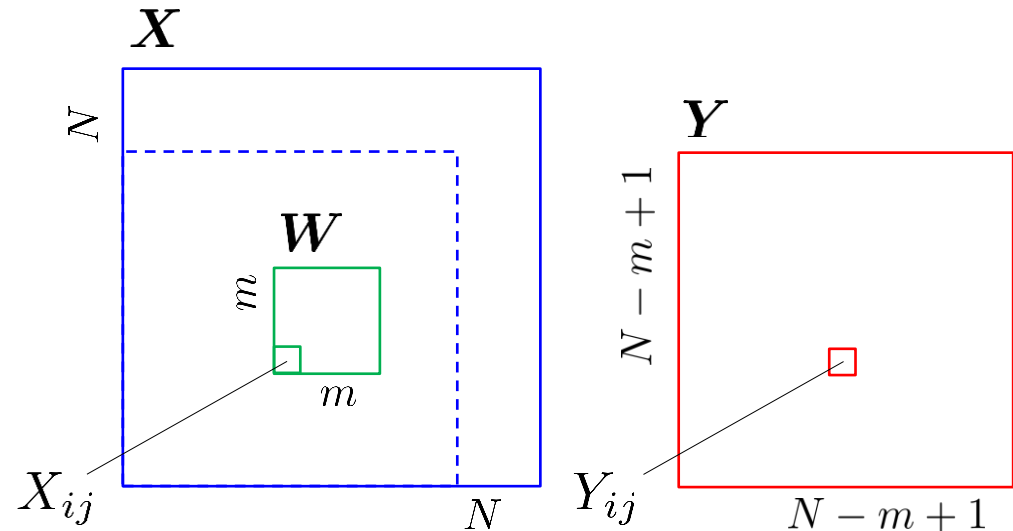
■ Gradient of convolutional layer

$$Y = (W * X)$$

$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Case 1:

$$\frac{\partial}{\partial W} E(Y) \quad \text{i.e. the 'end of the chain'}$$



$$\frac{\partial}{\partial W_{lk}} E(Y) = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E(Y)}{\partial Y_{ij}} \frac{\partial Y_{ij}}{\partial W_{lk}}$$

$\frac{\partial E_{ij}}{\partial Y_{ij}} := \frac{\partial E(Y)}{\partial Y_{ij}}$

i.e. the backpropagation component across Y_{ij}

$$\frac{\partial Y_{ij}}{\partial W_{lk}} = X_{(i+l)(j+k)}$$

$$\frac{\partial}{\partial W_{lk}} E(Y) = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \frac{\partial E(Y)}{\partial Y_{ij}} X_{(i+l)(j+k)}$$

Convolutional Layer Gradient

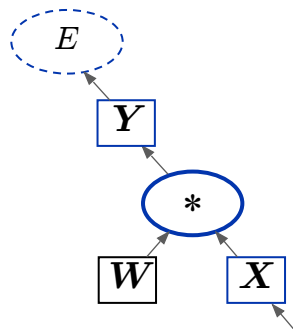
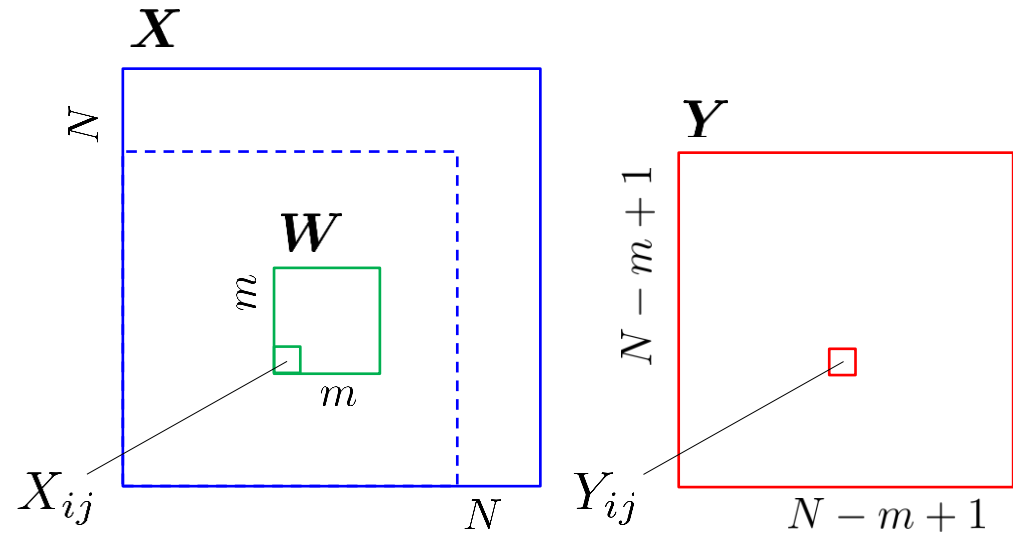
■ Gradient of convolutional layer

$$Y = (W * X)$$

$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Case 2:

$$\frac{\partial}{\partial \vartheta} E(Y) \quad \vartheta \neq W \text{ is a generic parameter on which } X \text{ depends}$$



Convolutional Layer Gradient

■ Gradient of convolutional layer

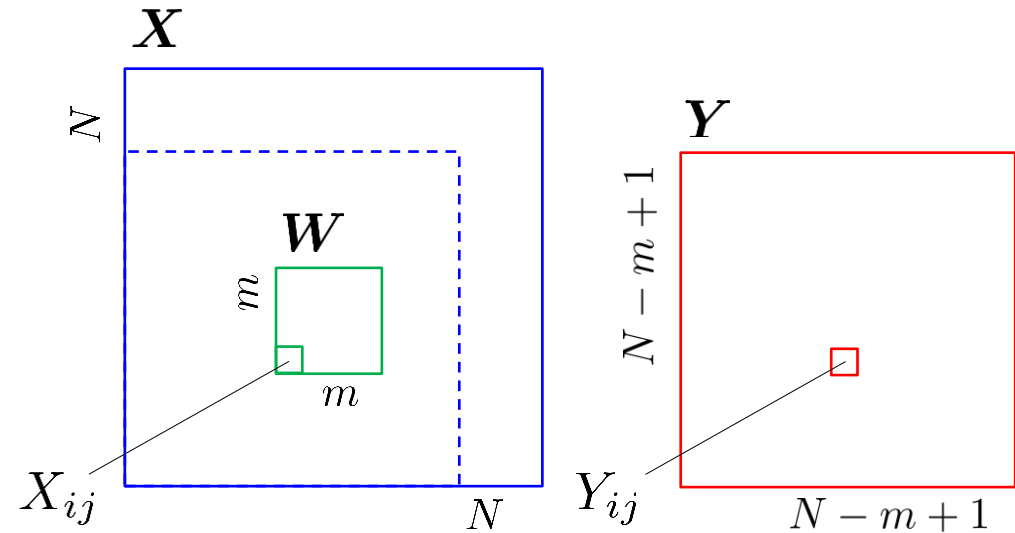
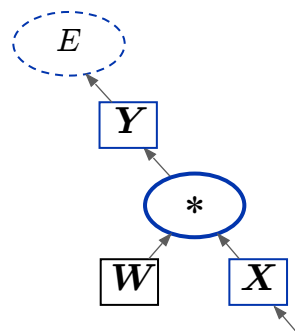
$$Y = (W * X)$$

$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Case 2:

$$\frac{\partial}{\partial \vartheta} E(Y) \quad \vartheta \neq W \text{ is a generic parameter on which } X \text{ depends}$$

$$\begin{aligned} \frac{\partial}{\partial \vartheta} E(Y) &= \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \partial E_{ij} \frac{\partial Y_{ij}}{\partial \vartheta} \\ &= \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \partial E_{ij} \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} \frac{\partial}{\partial \vartheta} X_{(i+a)(j+b)} \end{aligned}$$



This is inconvenient: the same X components appear multiple times - let's re-factor

Convolutional Layer Gradient

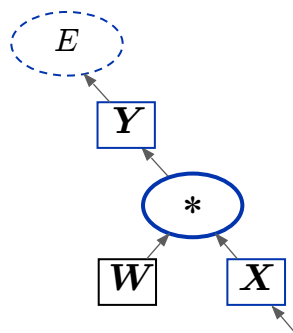
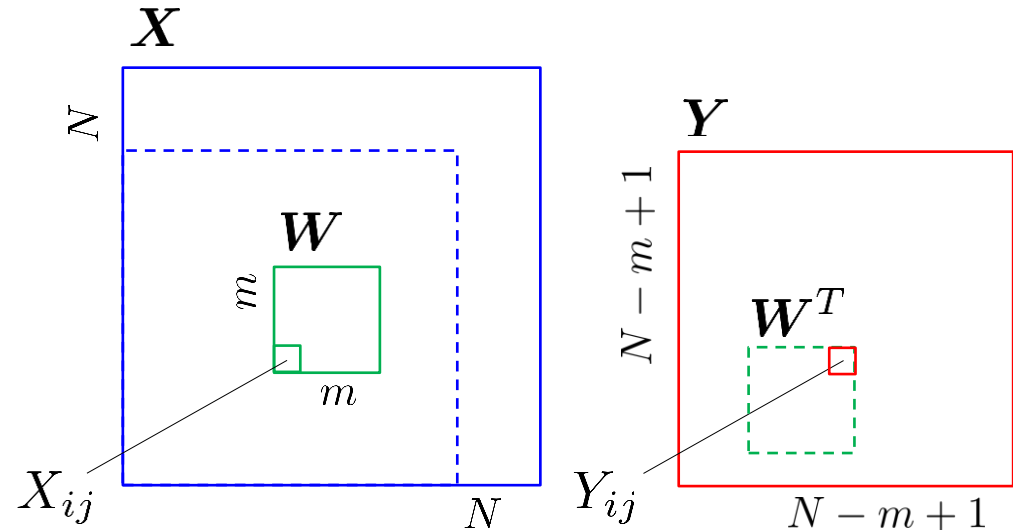
Gradient of convolutional layer

$$Y = (W * X)$$

$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Case 2:

$$\frac{\partial}{\partial \vartheta} E(Y) \quad \vartheta \neq W \text{ is a generic parameter on which } X \text{ depends}$$



$$\frac{\partial}{\partial \vartheta} E(Y) = \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \partial E_{ij} \frac{\partial Y_{ij}}{\partial \vartheta}$$

$$= \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \partial E_{ij} \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} \frac{\partial}{\partial \vartheta} X_{(i+a)(j+b)}$$

$$= \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \frac{\partial}{\partial \vartheta} X_{ij} \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ba} \frac{\partial E_{(i-a)(j-b)}}{\partial \vartheta} \quad \begin{array}{l} \swarrow (i-a), (j-b) \geq 0 \\ \searrow \text{note the inversion of indexes} \end{array}$$

Convolutional Layer Gradient

- **Gradient of ReLU** (see also Episode 1)

$$Y = \text{ReLU}(X)$$

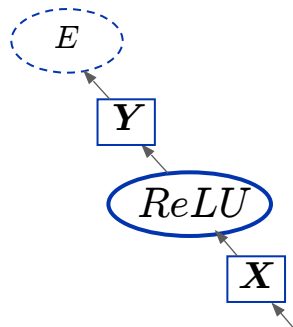
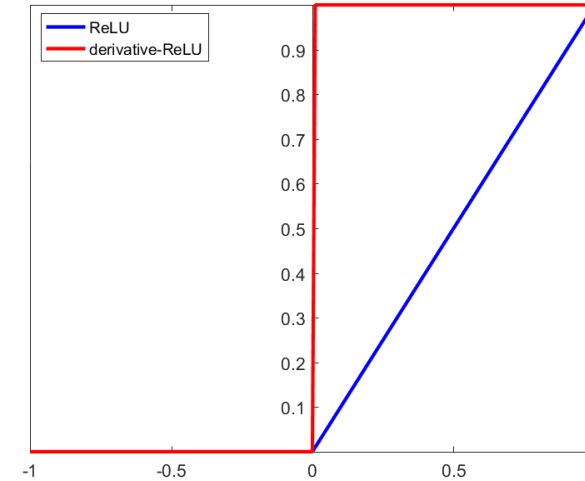
(ReLU has no parameters of its own)

$$\frac{\partial}{\partial x} \text{ReLU}(x) = \frac{\partial}{\partial x} \max(x, 0) \approx \text{step}(x)$$

So the gradient of ReLU acts like a 'switch'

When is it open?

Backpropagation alone 'does not know'



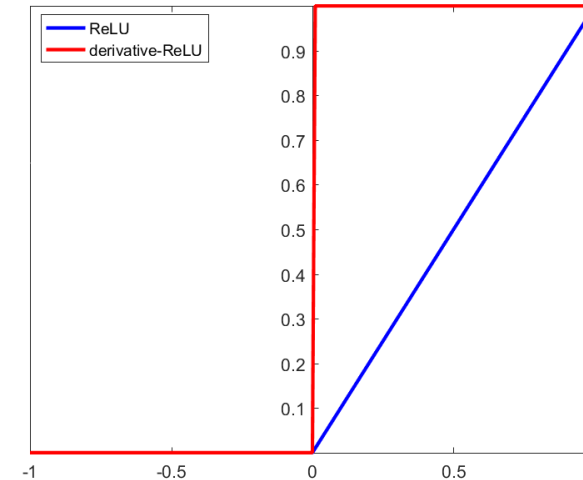
Convolutional Layer Gradient

■ Gradient of ReLU (see also Episode 1)

$$\mathbf{Y} = \text{ReLU}(\mathbf{X})$$

(ReLU has no parameters of its own)

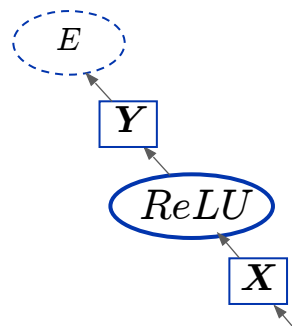
$$\frac{\partial}{\partial x} \text{ReLU}(x) = \frac{\partial}{\partial x} \max(x, 0) \approx \text{step}(x)$$



So the gradient of ReLU acts like a 'switch'

When is it open?

Backpropagation alone 'does not know'



$$\frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{Y})$$

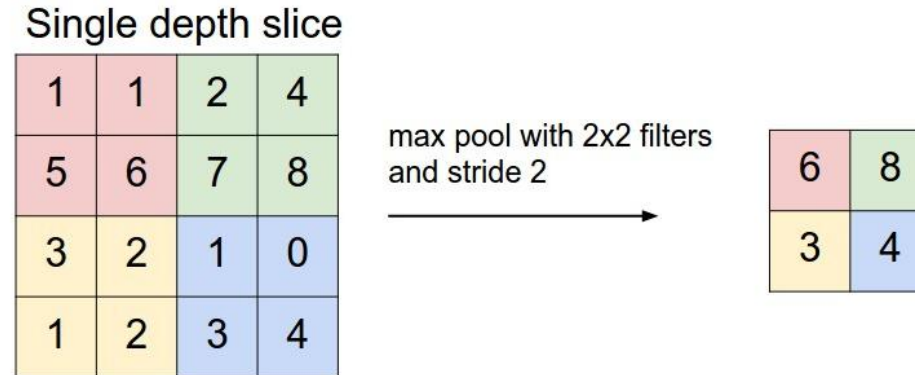
This is the gradient we want to compute

as we have to apply it to each specific data item $\left(\frac{\partial}{\partial \boldsymbol{\theta}} E(\mathbf{Y}) \right) (\mathbf{X}^{(i)})$

Moral: we need to perform one forward pass (i.e. activation) to decide which component Y_{ij} is open (i.e. /1) and which is not (i.e. /0)

Max Pooling Gradient

■ Gradient of Max Pooling



The gradient of max pooling acts as *multiplexer*

As with ReLU, one forward pass is required to determine which channel is selected

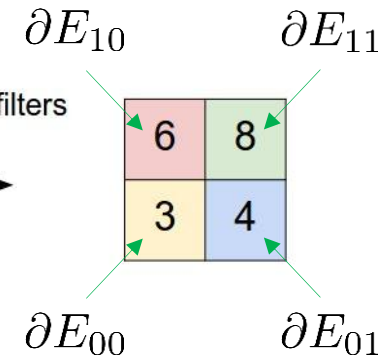
Max Pooling Gradient

■ Gradient of Max Pooling

Single depth slice

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters
and stride 2



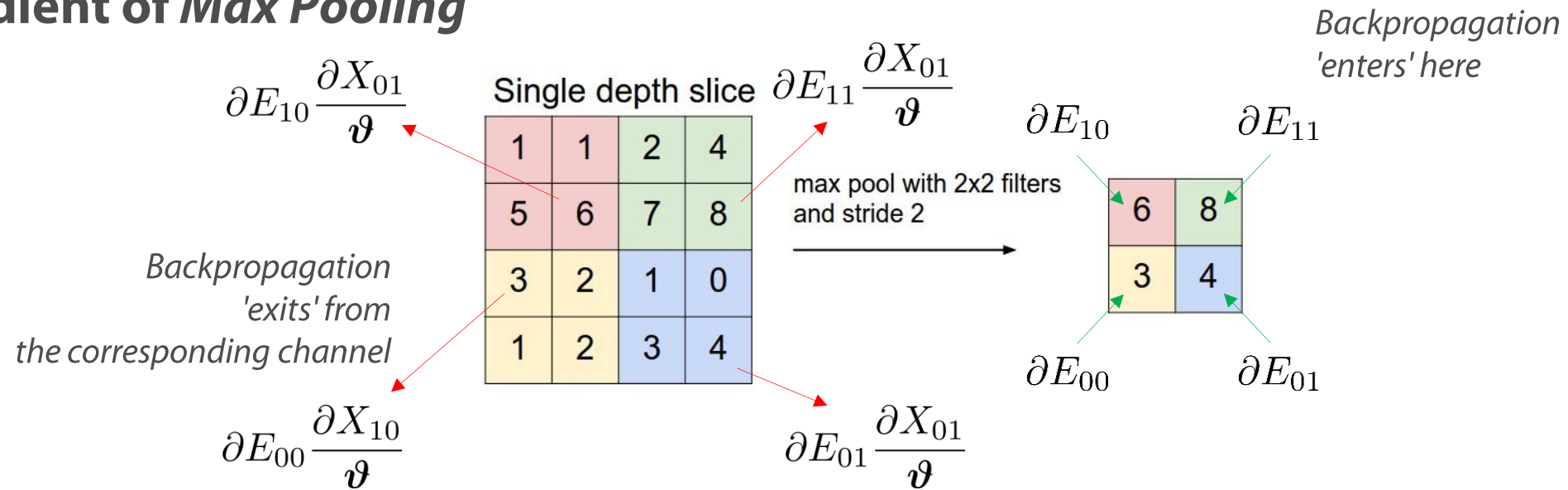
Backpropagation
'enters' here

The gradient of max pooling acts as *multiplexer*

As with ReLU, one forward pass is required to determine which channel is selected

Max Pooling Gradient

■ Gradient of Max Pooling



The gradient of max pooling acts as *multiplexer*

As with ReLU, one forward pass is required to determine which channel is selected

LRN Gradient

▪ Gradient of *Local Response Normalization*

$$Y_{ijl} := \frac{X_{ijl}}{\left(a + \alpha \sum_{k \in \text{Nbr}(i,j)} (X_{ijk})^2\right)^\beta}$$

where a, α, β are fixed hyperparameters

*This formula is quite inconvenient:
let's simplify....*

LRN Gradient

- **Gradient of Local Response Normalization**

$$Y_{ijl} := \frac{X_{ijl}}{\sum_{k=1}^h X_{ijk}}$$

i.e. plain, cross-map normalization

(simplified formula)

LRN Gradient

■ Gradient of Local Response Normalization

$$Y_{ijl} := \frac{X_{ijl}}{\sum_{k=1}^h X_{ijk}} \quad \text{i.e. plain, cross-map normalization}$$

i.e. the backpropagation component across Y_{ijl}

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} E(\mathbf{Y}) = \sum_{i,j} \sum_l \partial E_{ijl} \frac{\partial Y_{ijl}}{\partial \boldsymbol{\vartheta}} \quad \text{where } \partial E_{ijl} := \frac{\partial E(\mathbf{Y})}{\partial Y_{ijl}}$$

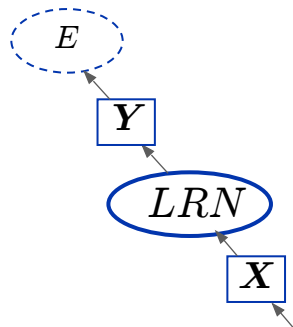
$$= \sum_{i,j} \sum_l \partial E_{ijl} \frac{\partial}{\partial \boldsymbol{\vartheta}} \frac{X_{ijl}}{\sum_k X_{ijk}}$$

$$= \sum_{i,j} \sum_l \partial E_{ijl} \left(\frac{1}{\sum_k X_{ijk}} \frac{\partial X_{ijl}}{\partial \boldsymbol{\vartheta}} - \frac{X_{ijl}}{(\sum_k X_{ijk})^2} \sum_k \frac{\partial X_{ijk}}{\partial \boldsymbol{\vartheta}} \right)$$

$$= \sum_{i,j} \sum_l \partial E_{ijl} \left(\frac{1}{c} \frac{\partial X_{ijl}}{\partial \boldsymbol{\vartheta}} - \frac{Y_{ijl}}{c} \sum_k \frac{\partial X_{ijk}}{\partial \boldsymbol{\vartheta}} \right)$$

where

$$c := \sum_k X_{ijk}$$



LRN Gradient

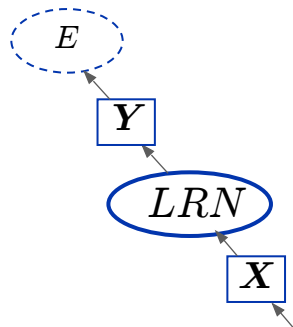
▪ Gradient of Local Response Normalization

$$Y_{ijl} := \frac{X_{ijl}}{\sum_{k=1}^h X_{ijk}} \quad \text{i.e. plain, cross-map normalization}$$

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} E(\mathbf{Y}) = \sum_{i,j} \sum_l \partial E_{ijl} \left(\frac{1}{c} \frac{\partial X_{ijl}}{\partial \boldsymbol{\vartheta}} - \frac{Y_{ijl}}{c} \sum_k \frac{\partial X_{ijk}}{\partial \boldsymbol{\vartheta}} \right)$$

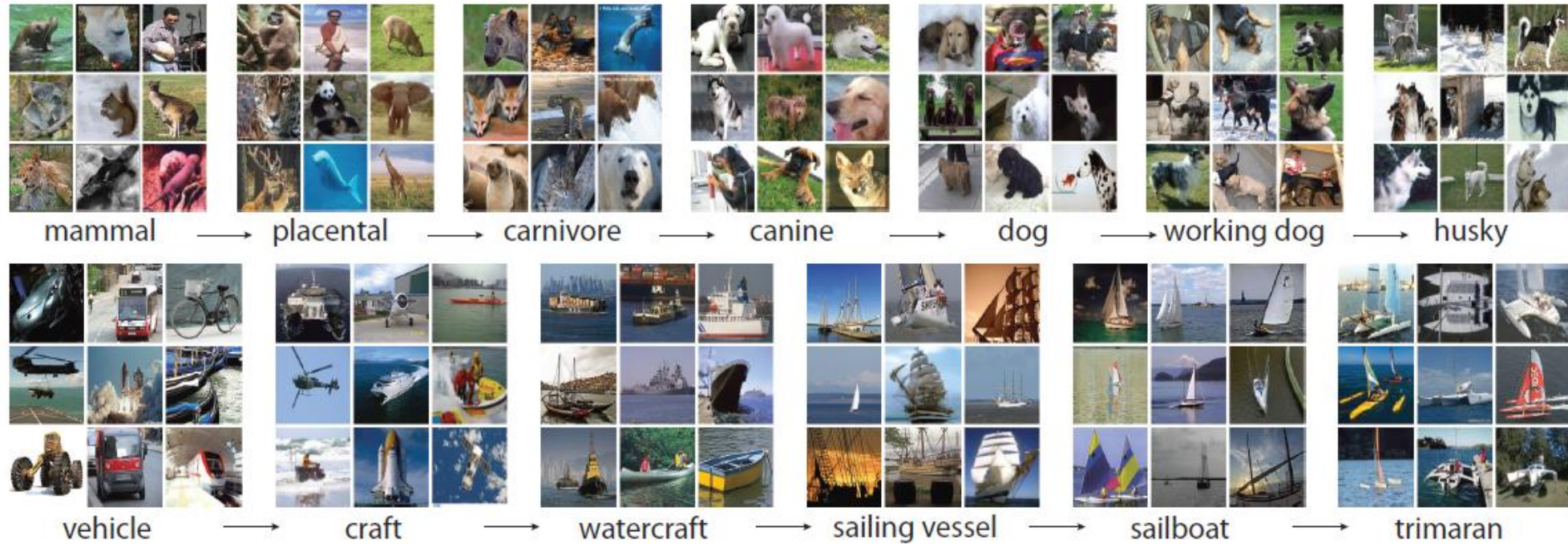
This is inconvenient: the same X components appear multiple times - let's re-factor

$$= \sum_{i,j} \sum_l \left(\frac{\partial E_{ijl}}{c} - \sum_k \left(Y_{ijk} \frac{\partial E_{ijk}}{c} \right) \right) \frac{\partial X_{ijl}}{\partial \boldsymbol{\vartheta}}$$



ImageNet Challenge

- The ImageNet Large Scale Visual Recognition Challenge



1,461,406 full resolution images

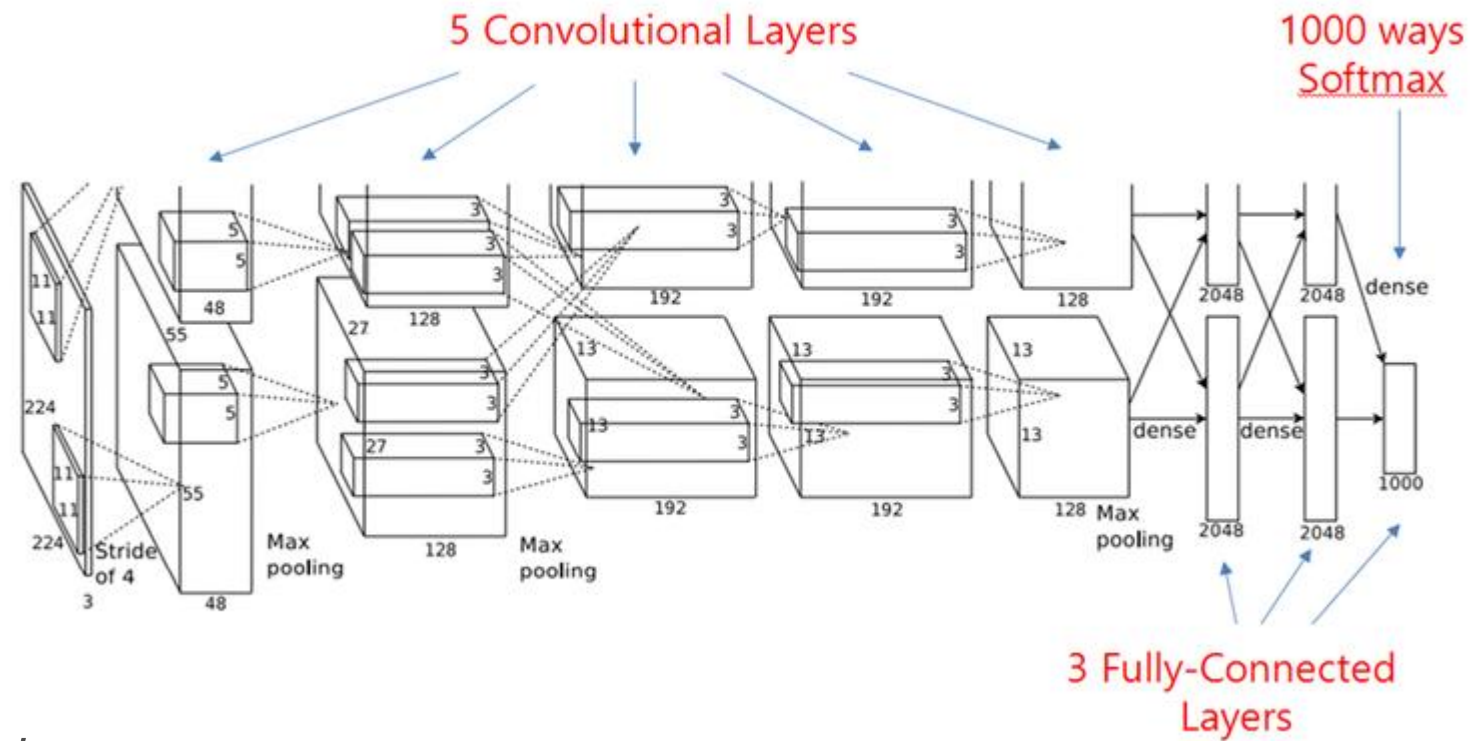
Complex and multiple textual annotation,
hierarchy of 1000 object classes along several dimensions

The image classification challenge is run annually since 2010

[figures from www.nvidia.com]

AlexNet (Krizhevsky, Sutskever & Hinton, 2012)

"The Mother of all DCNNs"



Trained with *batch gradient descent*

- the final supervised training set contained 15M images
- training was performed on two NVIDIA GTX 580 GPUs for six days

[image from <https://world4jason.gitbooks.io/research-log/content/deepLearning/CNN/Model%20%20ImgNet/alexnet/alexnet.html>]

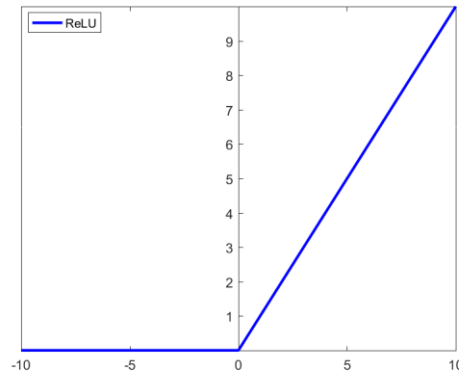
Deep Convolutional Neural Networks (DCNN)

■ AlexNet

Why ReLU and not another non-linearity?

Because it is much faster to train.

$$y = \max(0, x)$$



ReLU

Image from [Krizhevsky, Sutskever & Hinton, 2012]

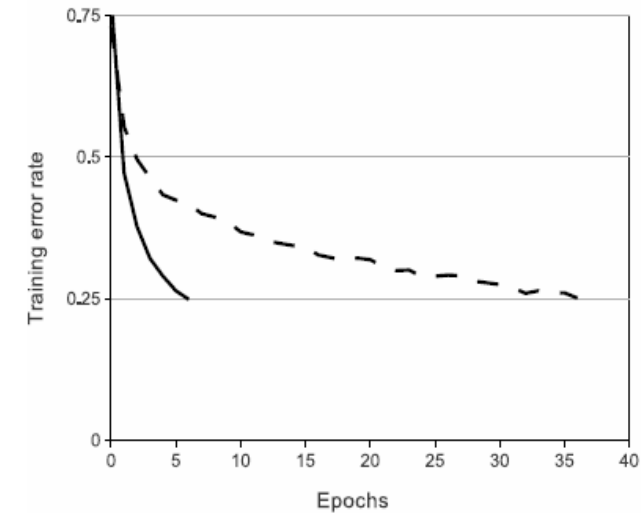


Figure 1: A four-layer convolutional neural network with ReLUs (**solid line**) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons (**dashed line**). The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.