Università degli
Studi di Pavia

Deep Learning

06- Deep Convolutional Neural Networks

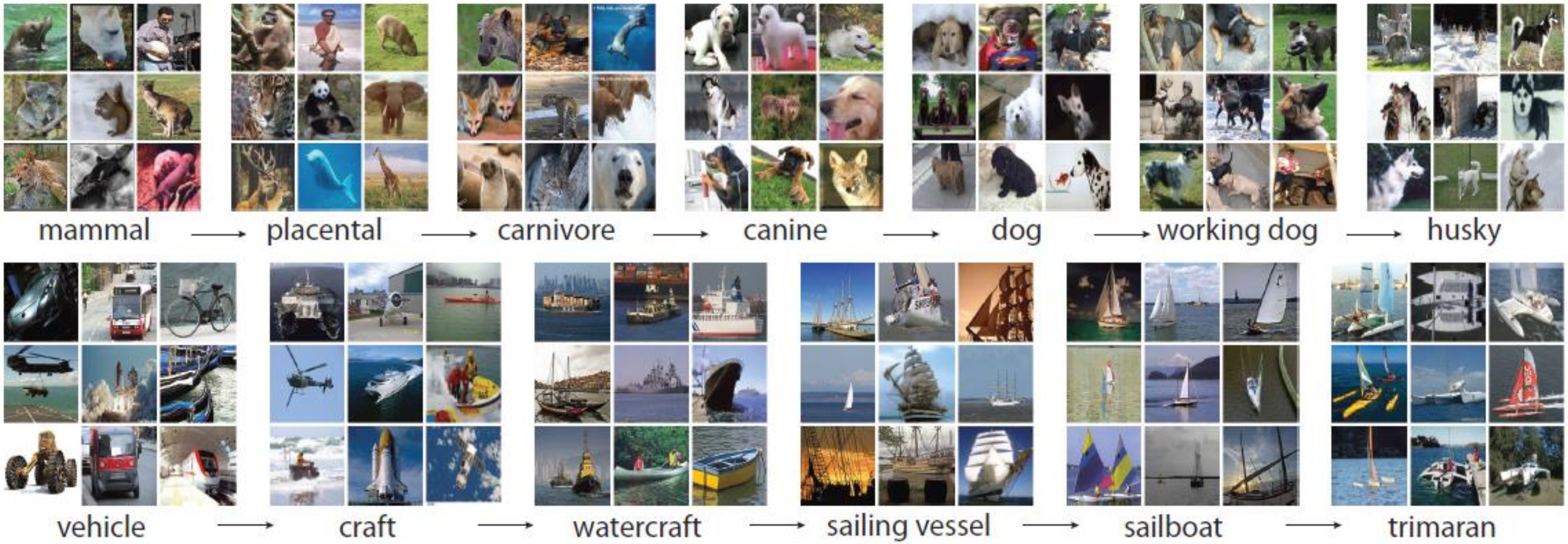
Marco Piastra & Andrea Pedrini(*)

(*) Dipartimento di Matematica F. Casorati

This presentation can be downloaded at:
http://vision.unipv.it/DL

ImageNet Challenge

- The ImageNet Large Scale Visual Recognition Challenge



1,461,406 full resolution images

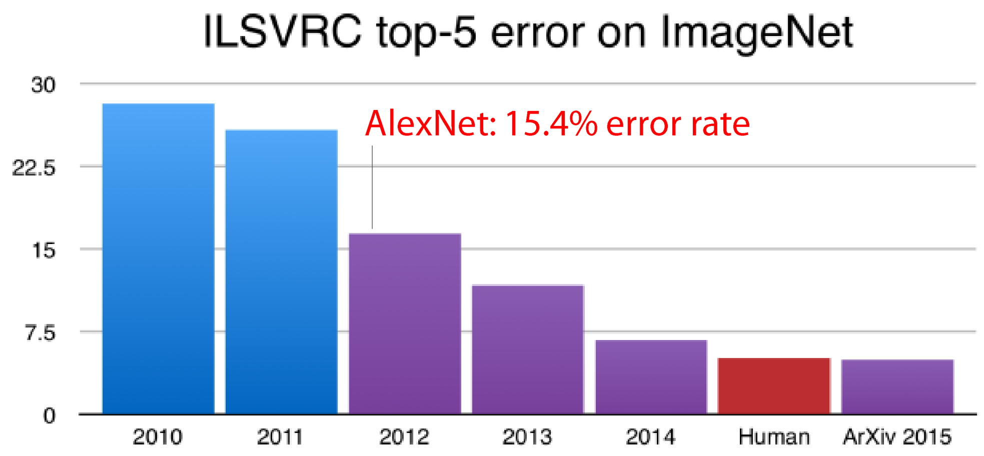
Complex and multiple textual annotation,
hierarchy of 1000 object classes along several dimensions

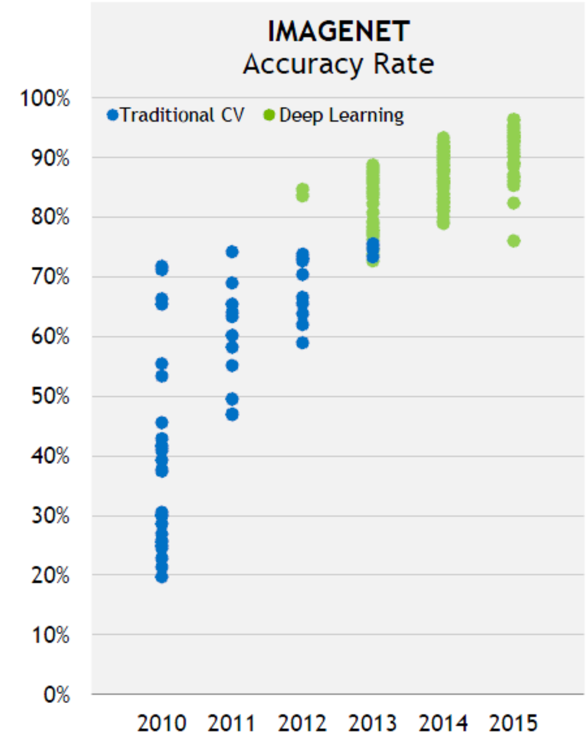
The image classification challenge is run annually since 2010

[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]

Deep Convolutional Neural Network (DCNN)

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

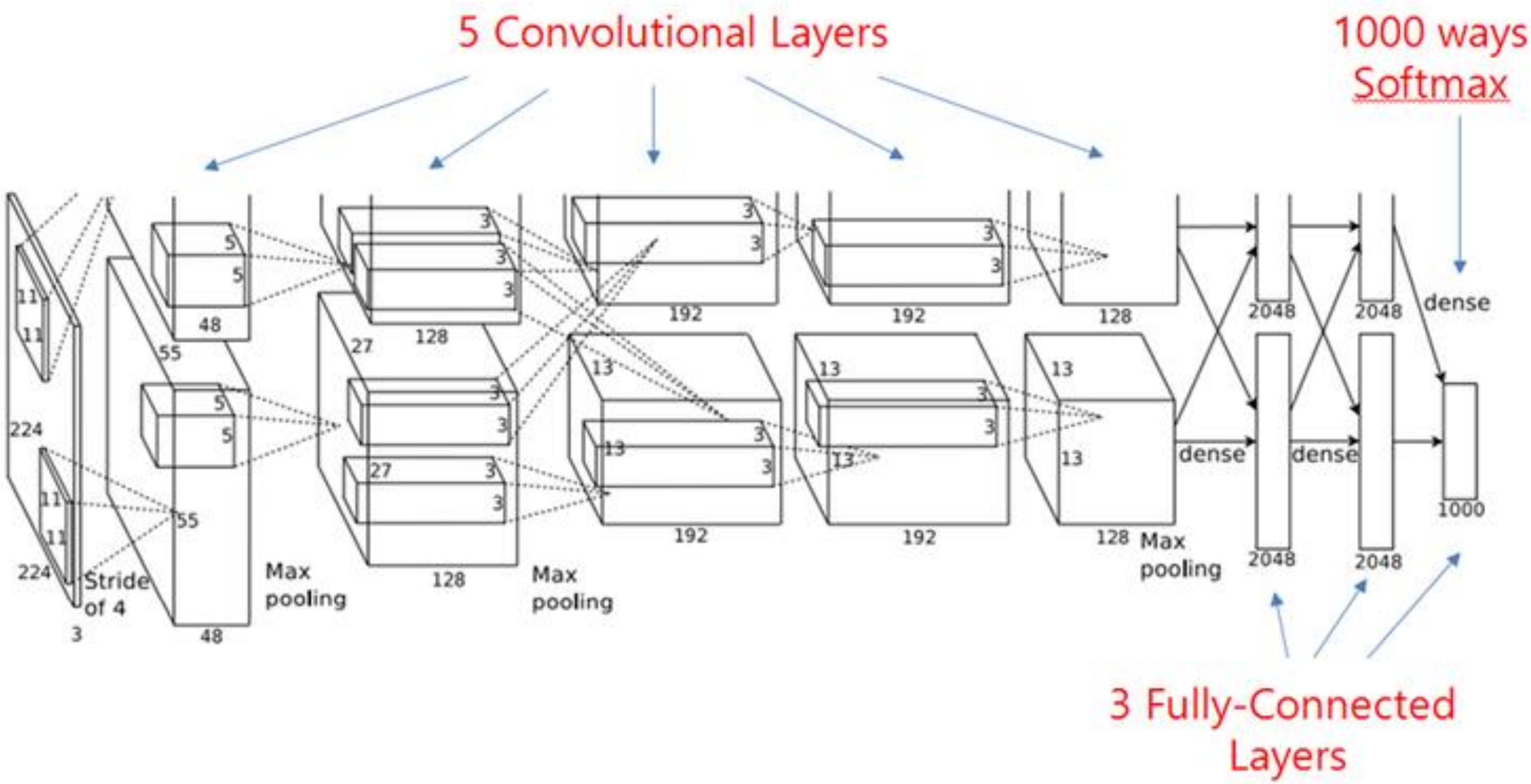


Image from [Krizhevsky, Sutskever & Hinton, 2012]

Deep Convolutional Neural Network (DCNN)

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

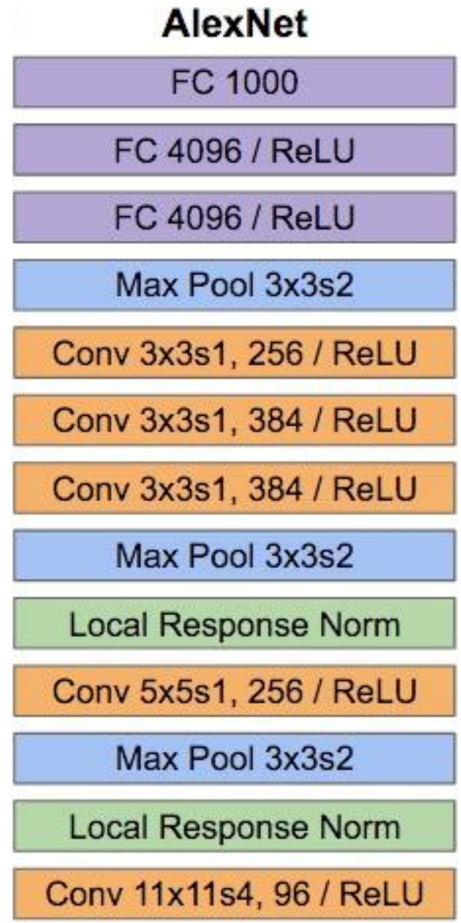
This layer is ***Softmax***

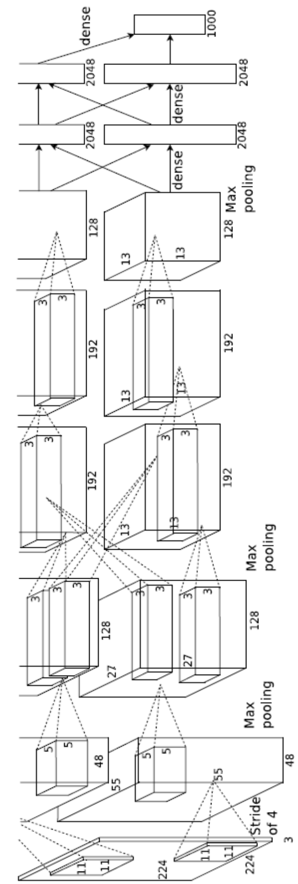
These layers are ***fully connected (FC)*** FF neural networks

This layer is ***Max Pooling***

These layers are ***convolutional*** with ***ReLU*** as non-linearity

This layer is ***Local Response Normalization (LRN)***





DCNN Building Blocks (layerwise)

Convolutional Layer

- ***Convolution operation***

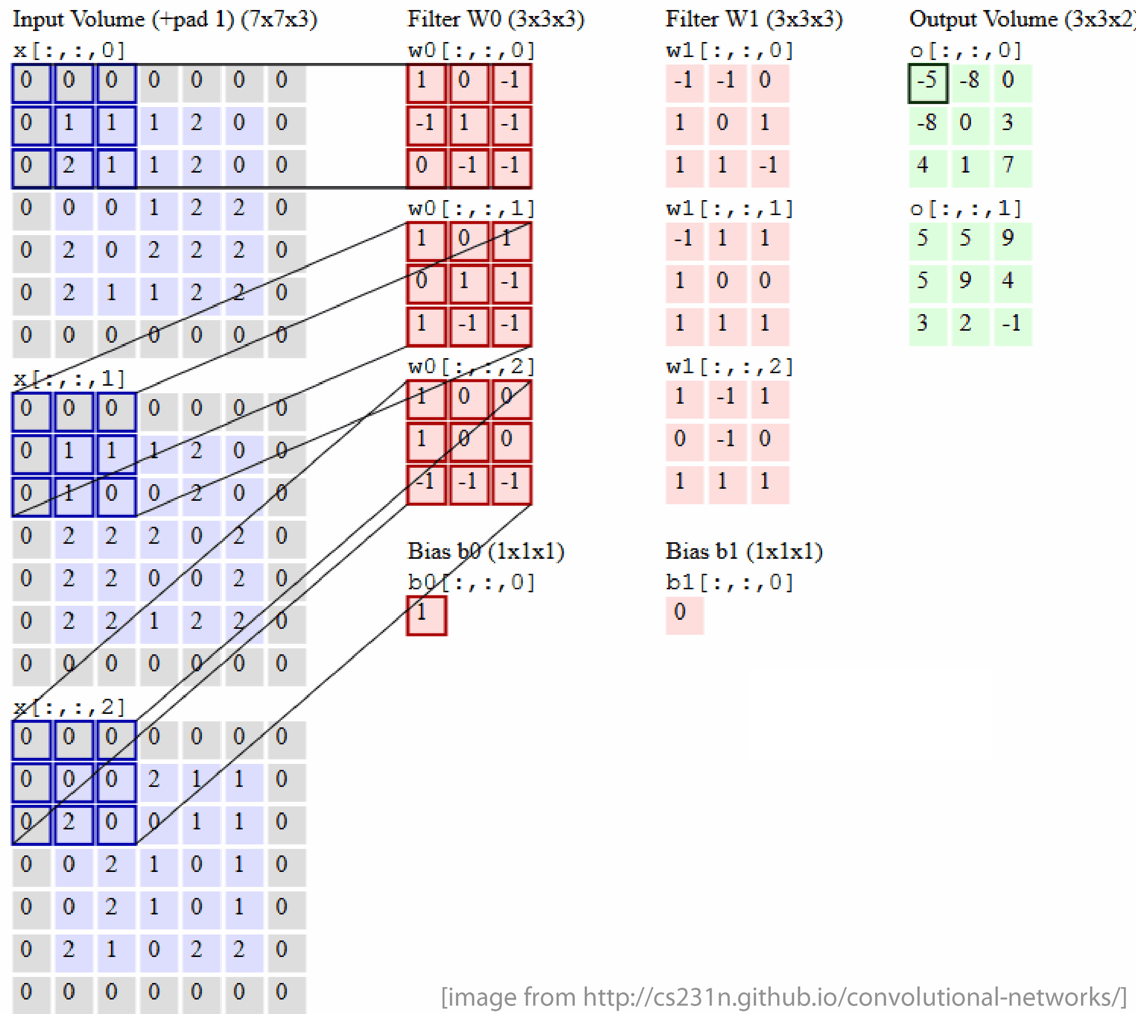
A ***convolution filter***
is a square (or cubic) matrix

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/]

Convolution operations (on images)

A ***convolution filter***
is a square (or cubic) matrix

In symbols

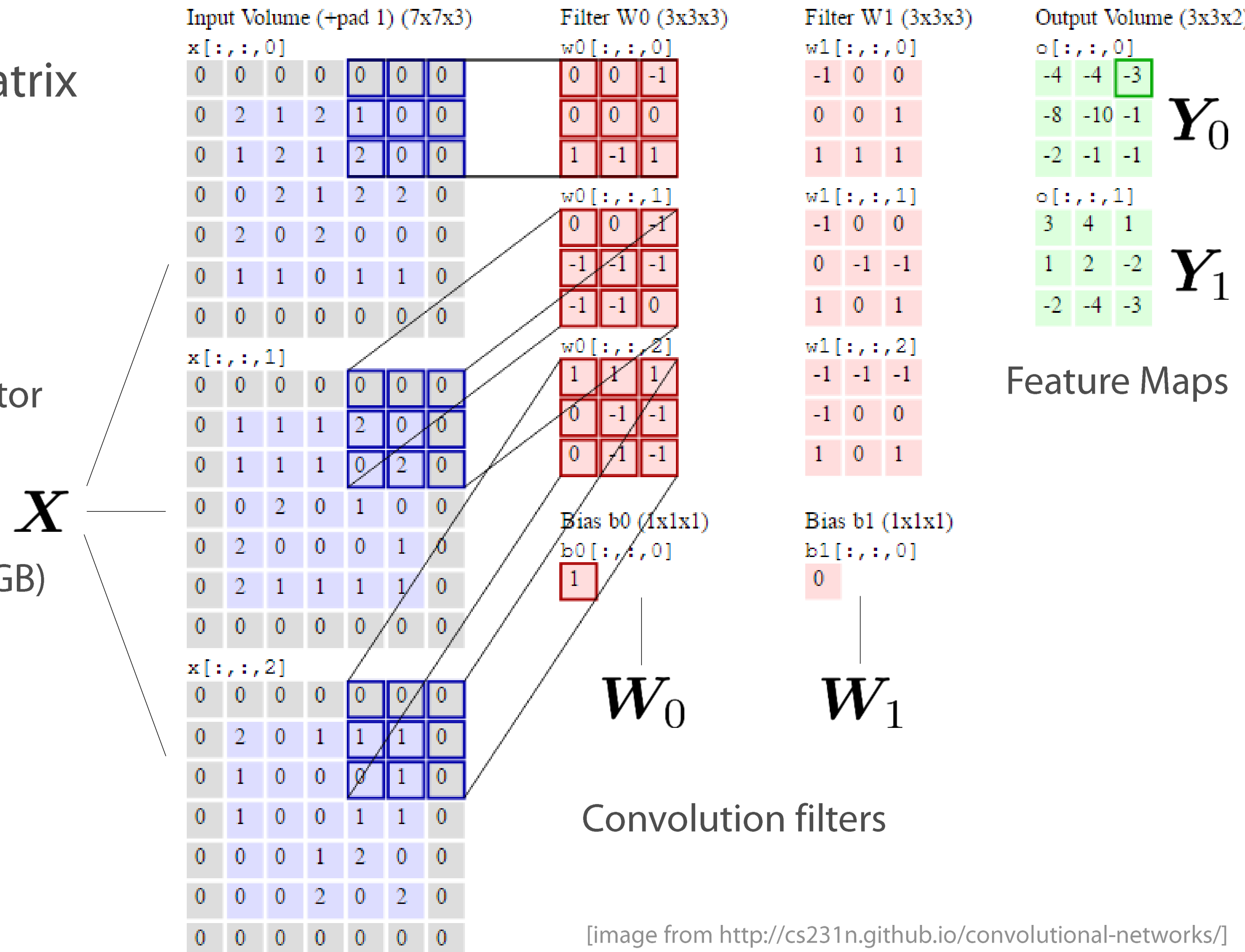
$$\boldsymbol{Y}_i := \boldsymbol{W}_i * \boldsymbol{X}$$

i-th feature map

convolution operator

where:

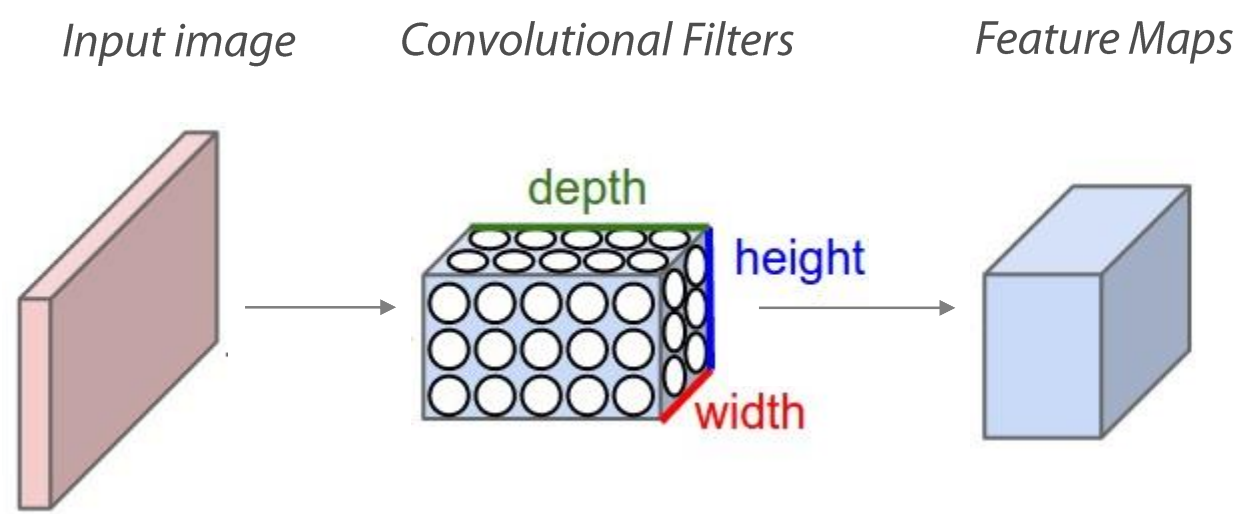
Input image (e.g. RGB)



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

Convolutional Layer

- **Convolution network (first layer)**



$\boldsymbol{X}$ $[\boldsymbol{W}_0, \cdots, \boldsymbol{W}_h]$ $[\boldsymbol{Y}_0, \cdots, \boldsymbol{Y}_h]$

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

Convolutional Layer

- ***Convolution operation with non-linearity***

The linear form for convolution

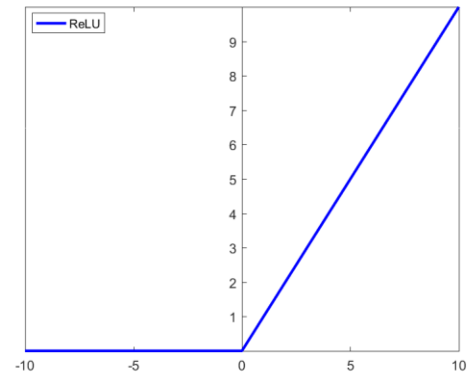
$$\boldsymbol{Y}_i := \boldsymbol{W}_i * \boldsymbol{X}$$

in actual networks is composed with a non-linearity

$$\boldsymbol{Y}_i := \mathrm{ReLU}(\boldsymbol{W}_i * \boldsymbol{X})$$

Applied elementwise to all matrix components

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



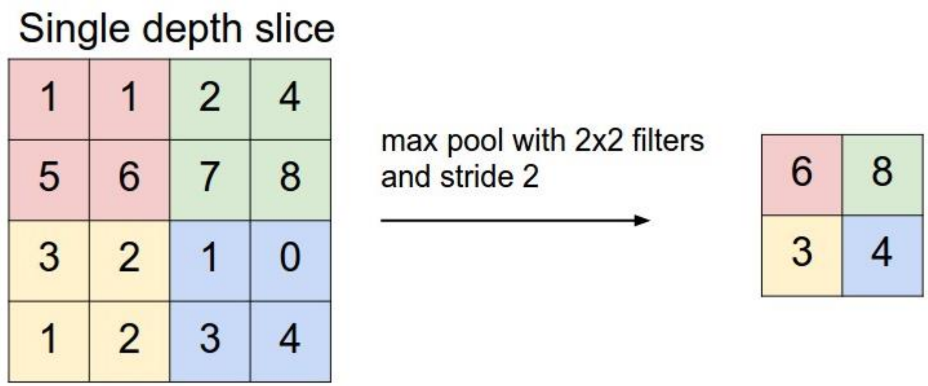
ReLU

Why ReLU?
To be seen later on, when discussing training

Max Pooling Layer

- ***Max Pooling operation***

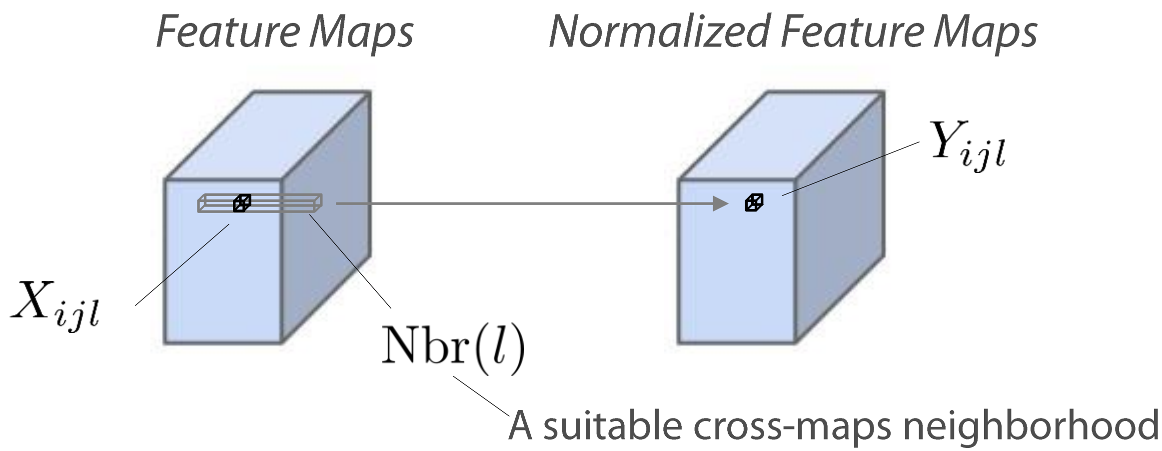
  Returns the maximum value in a pre-defined region of its input



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)

$$[\boldsymbol{X}_0, \cdots, \boldsymbol{X}_h] \qquad [\boldsymbol{Y}_0, \cdots, \boldsymbol{Y}_h]$$

$$\boldsymbol{X}_l := [X_{ijl}] \qquad \boldsymbol{Y}_l := [Y_{ijl}]$$

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

where a, α, β are fixed *hyperparameters*

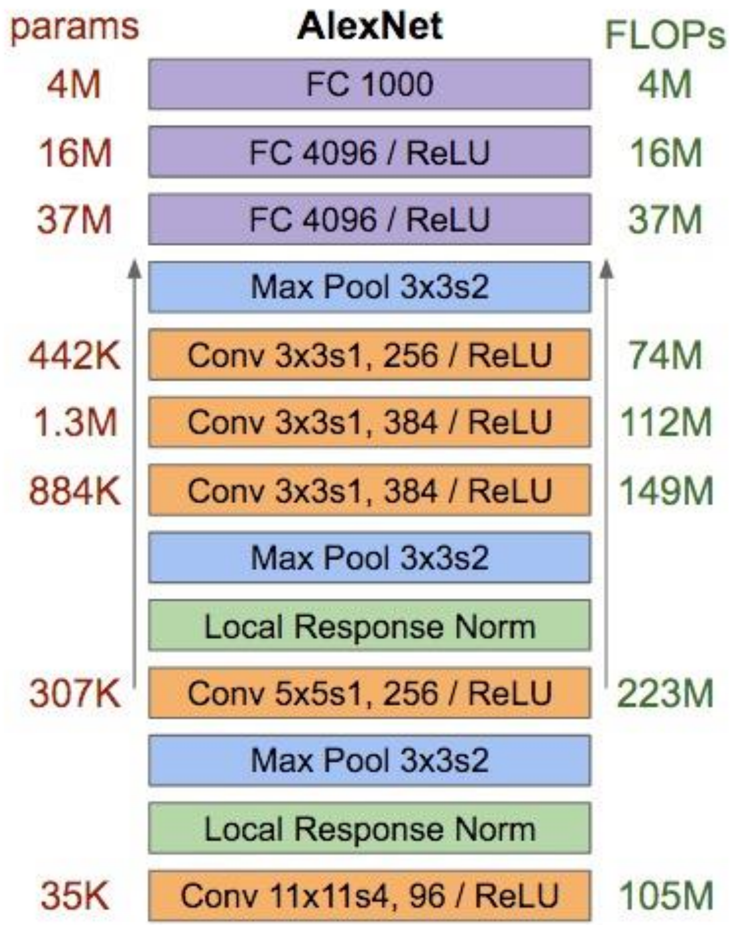
AlexNet Architecture

- ***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

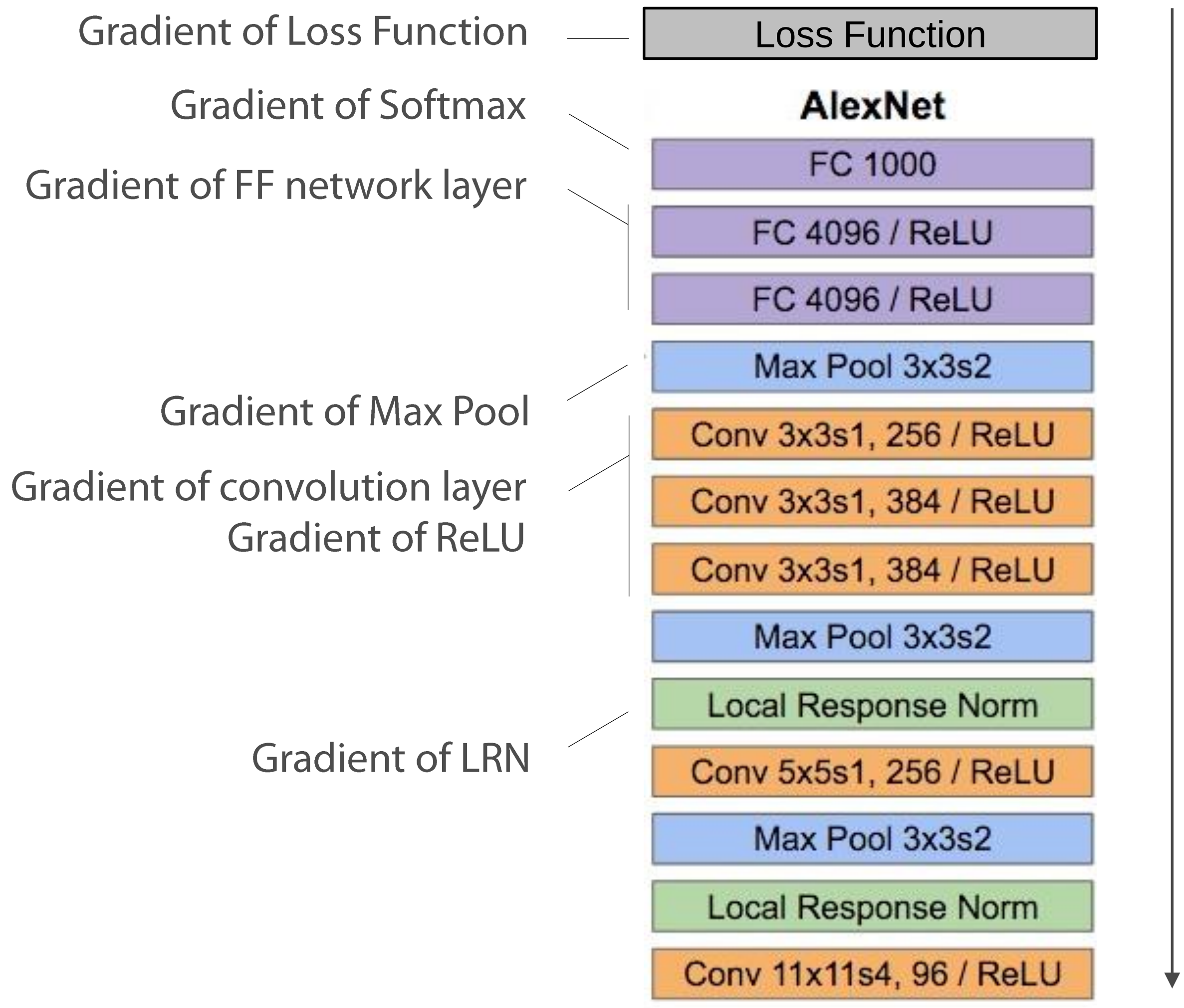
Totals:

- around 60M parameters
- around 837M FLOPs for a single pass



AlexNet Gradient

- ***Computing gradients (backward propagation)***



Convolutional Layer Gradient

- **Gradient of *convolutional layer***

  Define

  $$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

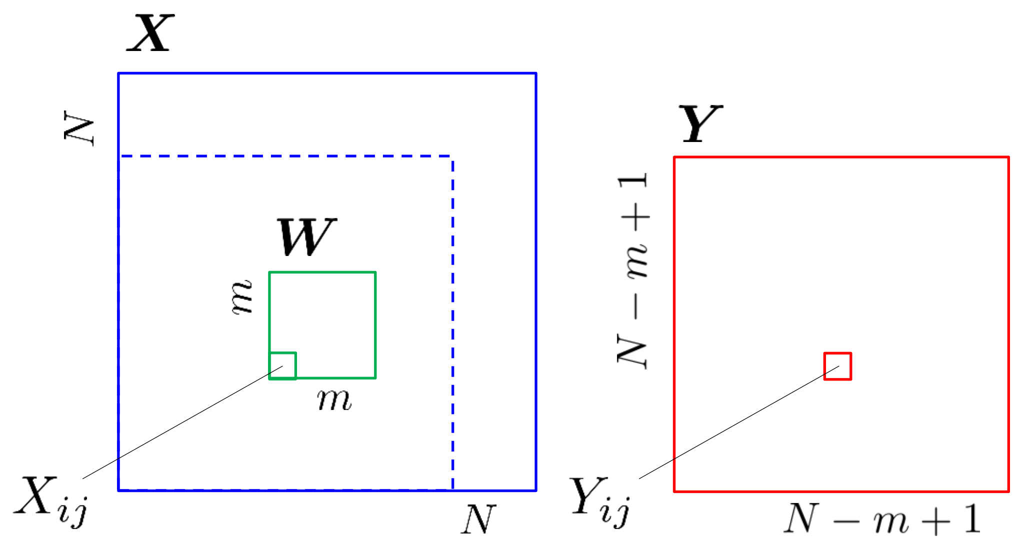
  and, for convenience, assume that

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

  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 0,0

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

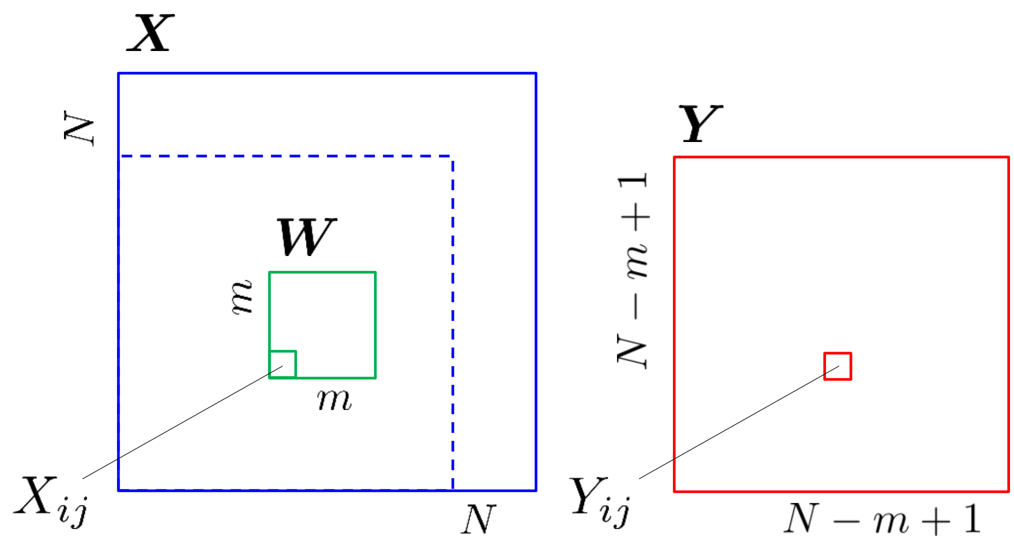
- **Gradient of *convolutional layer***

$$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

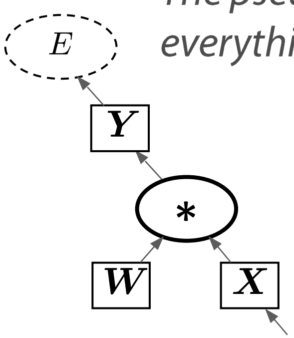
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Consider the pseudo-graph

The flow is bottom-up, in this example



The pseudo-node E represents everything that stands above



Another subgraph stands below (not represented)

Convolutional Layer Gradient

- **Gradient of *convolutional layer***

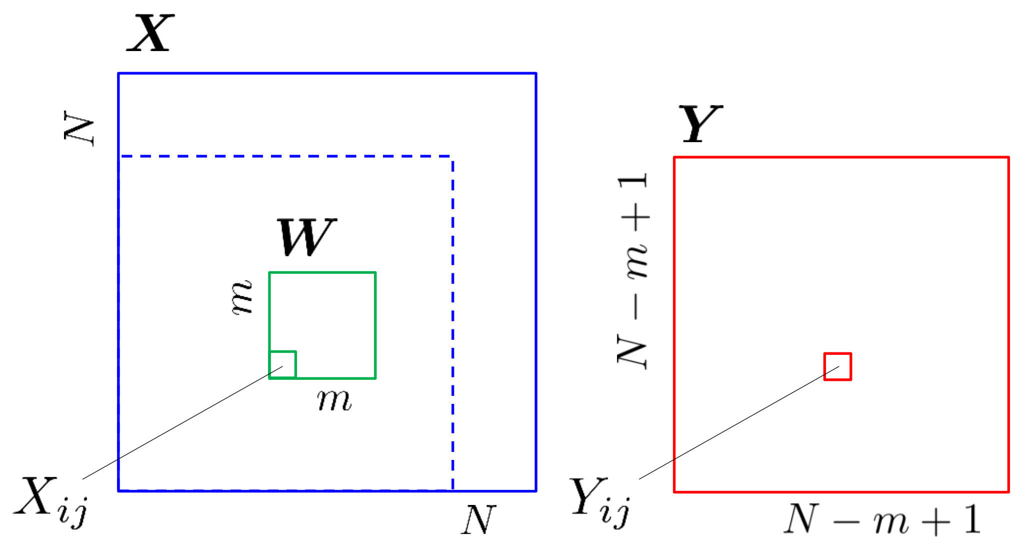
$$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

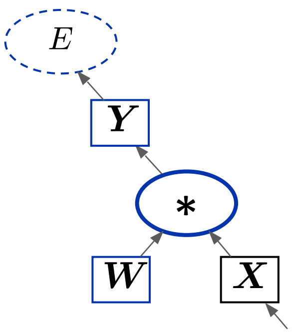
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Case 1:

$$\frac{\partial}{\partial \boldsymbol{W}} E(\boldsymbol{Y})$$

i.e. the 'end of the chain'



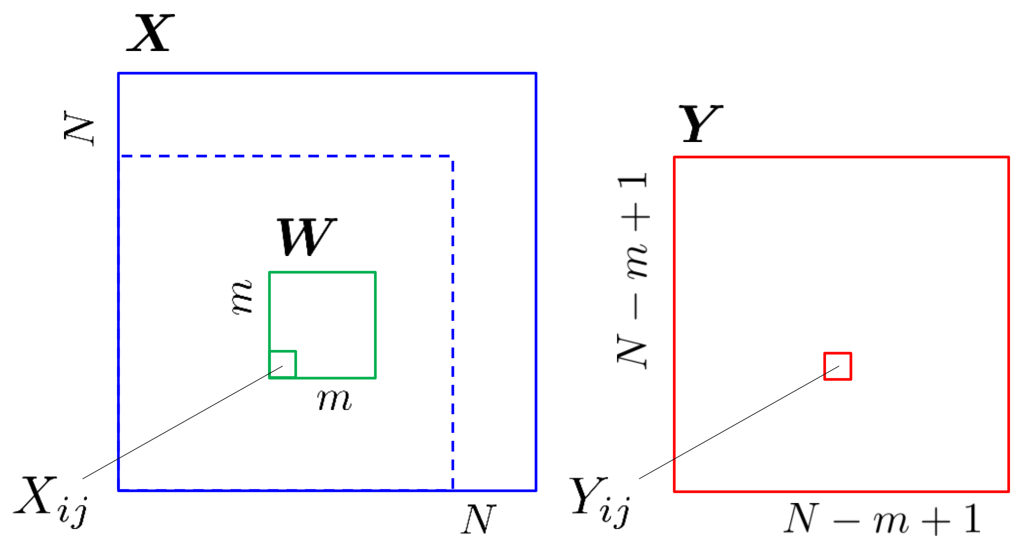


Convolutional Layer Gradient

- **Gradient of *convolutional layer***

$$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

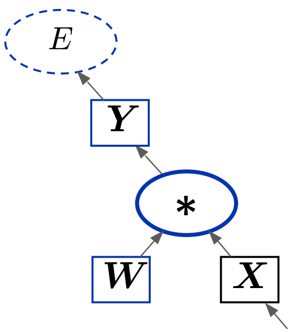
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$



Case 1:

$$\frac{\partial}{\partial \boldsymbol{W}} E(\boldsymbol{Y})$$

i.e. the 'end of the chain'



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

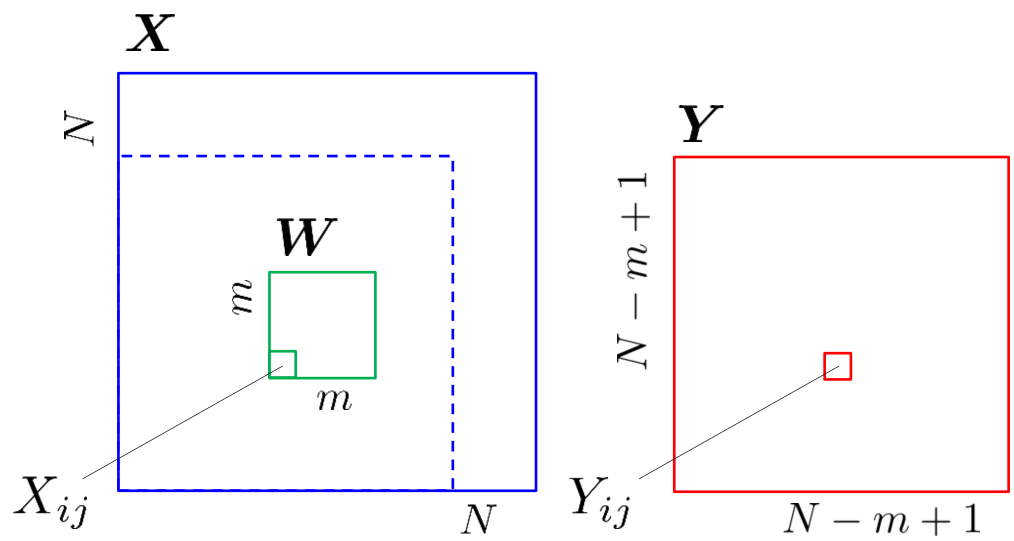
by applying the chain rule
in extended version (see also Episode 1)

Convolutional Layer Gradient

- **Gradient of *convolutional layer***

$$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

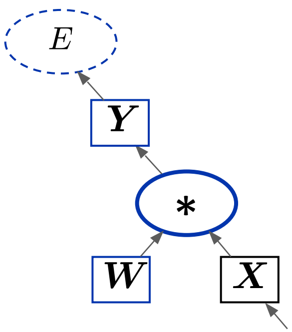
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$



Case 1:

$$\frac{\partial}{\partial \boldsymbol{W}} E(\boldsymbol{Y})$$

i.e. the 'end of the chain'



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

$$\partial E_{ij} := \frac{\partial E(\boldsymbol{Y})}{\partial Y_{ij}} \qquad \frac{\partial Y_{ij}}{\partial W_{lk}} = X_{(i+l)(j+k)}$$

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

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

Convolutional Layer Gradient

- **Gradient of *convolutional layer***

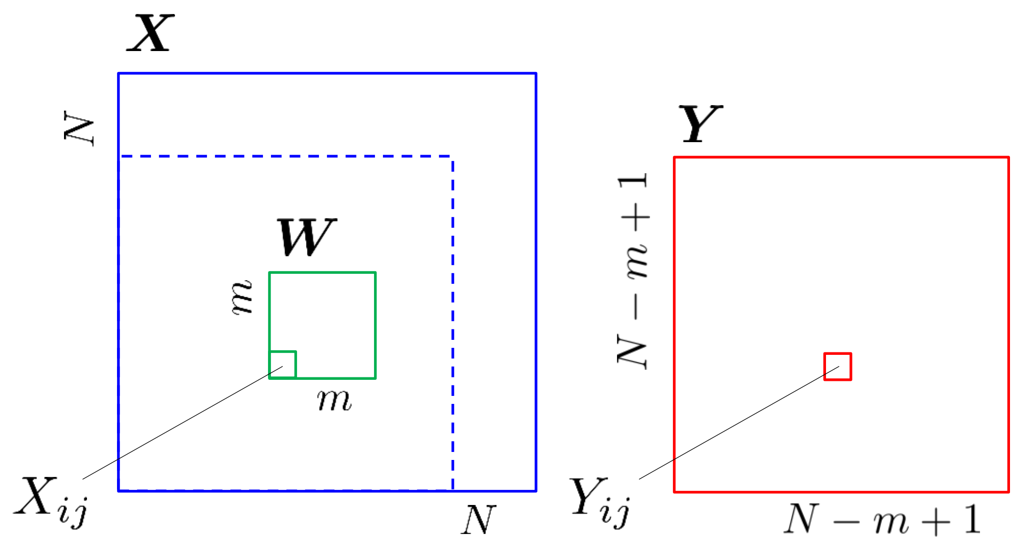
$$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

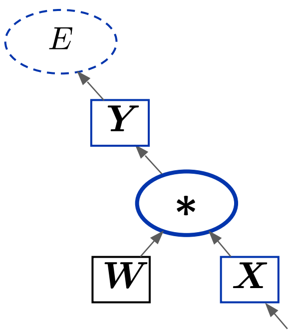
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$

Case 2:

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} E(\boldsymbol{Y})$$

$\boldsymbol{\vartheta} \neq \boldsymbol{W}$ is a generic parameter on which $\boldsymbol{X}$ depends



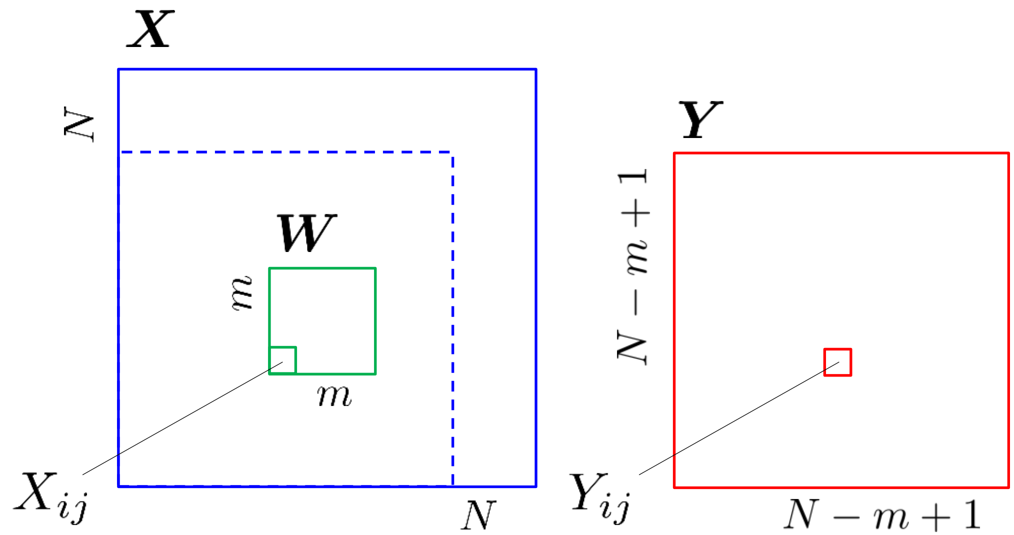


Convolutional Layer Gradient

- **Gradient of *convolutional layer***

$$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

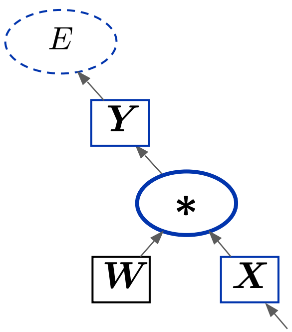
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$



Case 2:

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} E(\boldsymbol{Y})$$

$\boldsymbol{\vartheta} \neq \boldsymbol{W}$ is a generic parameter on which $\boldsymbol{X}$ depends



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

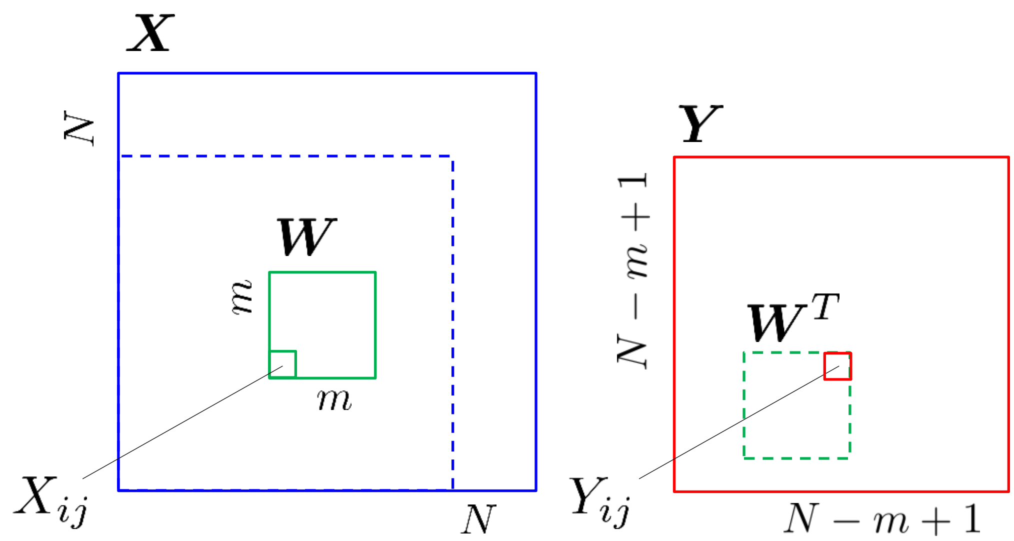
$$= \sum_{i=0}^{N-m} \sum_{j=0}^{N-m} \partial E_{ij} \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} \frac{\partial}{\partial \boldsymbol{\vartheta}} X_{(i+a)(j+b)}$$

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

- **Gradient of *convolutional layer***

$$\boldsymbol{Y} = (\boldsymbol{W} * \boldsymbol{X})$$

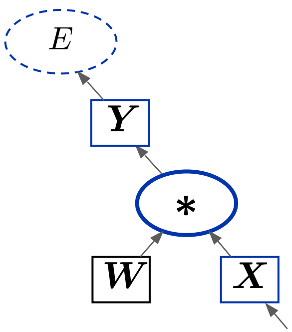
$$Y_{ij} = \sum_{a=0}^{m-1} \sum_{b=0}^{m-1} W_{ab} X_{(i+a)(j+b)}$$



Case 2:

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} E(\boldsymbol{Y})$$

$\boldsymbol{\vartheta} \neq \boldsymbol{W}$ is a generic parameter on which $\boldsymbol{X}$ depends



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

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

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

$(i-a),(j-b) \geq 0$

note the inversion of indexes

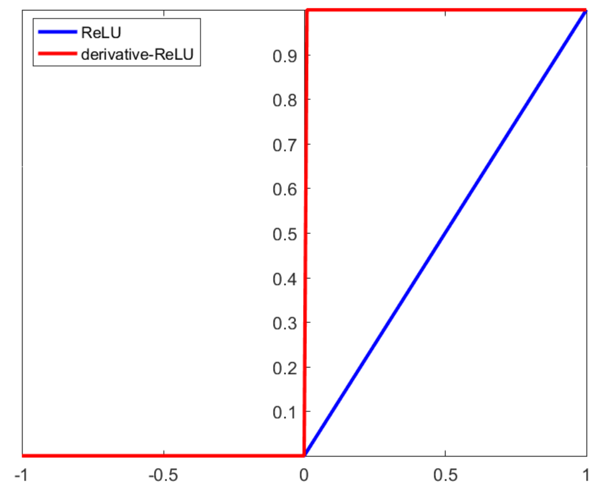
Convolutional Layer Gradient

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

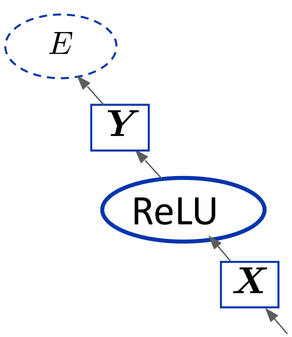
$$\boldsymbol{Y} = \mathrm{ReLU}(\boldsymbol{X})$$

(ReLU has no parameters of its own)

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



So the gradient of ReLU acts like a 'switch'
When is it open? *Backpropagation alone 'does not know'*



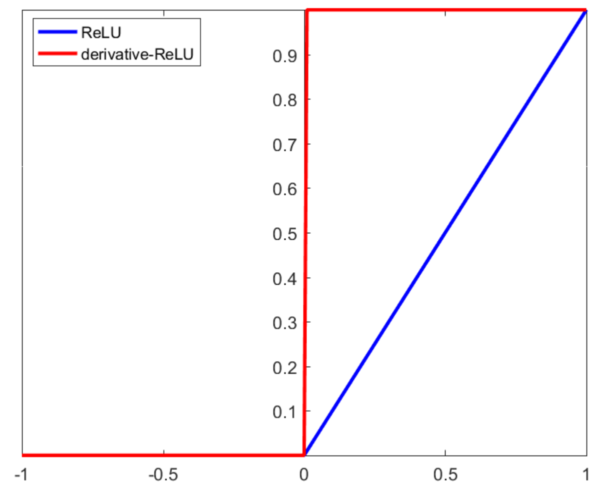
Convolutional Layer Gradient

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

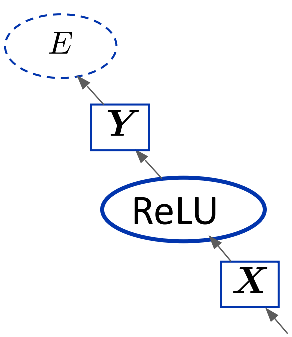
$$\boldsymbol{Y} = \mathrm{ReLU}(\boldsymbol{X})$$

(ReLU has no parameters of its own)

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



So the gradient of ReLU acts like a 'switch'
When is it open? *Backpropagation alone 'does not know'*



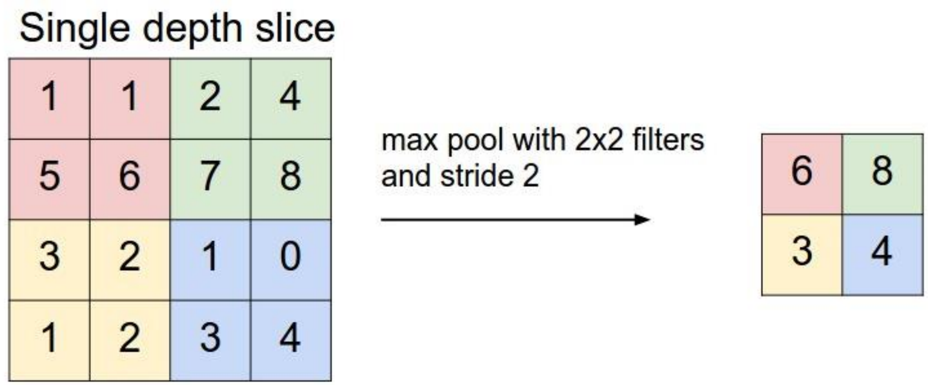
$$\frac{\partial}{\partial \boldsymbol{\vartheta}}E(\boldsymbol{Y})$$ *This is the gradient we want to compute*

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

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***

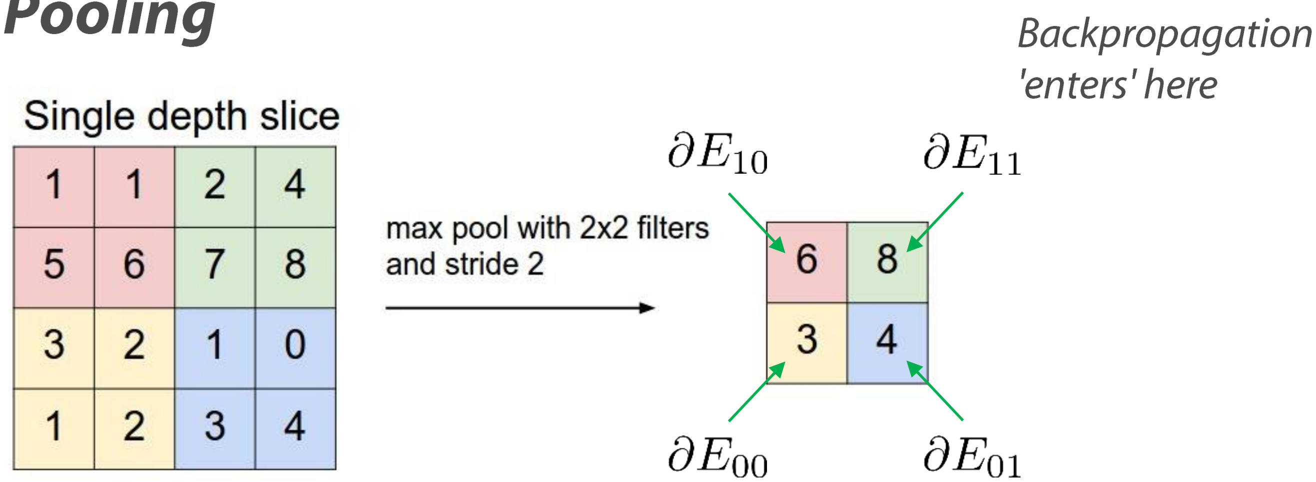


The gradient of max pooling acts as *multiplexer*

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

Max Pooling Gradient

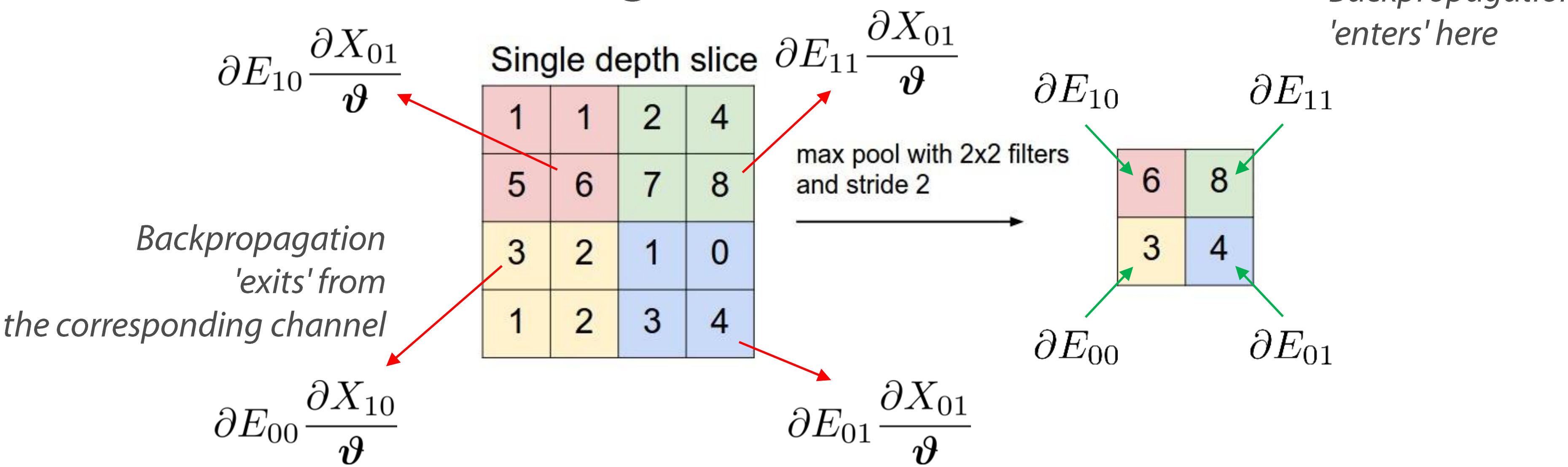
- **Gradient of *Max Pooling***



The gradient of max pooling acts as *multiplexer*

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

- **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***

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

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***

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

i.e. plain, cross-map normalization

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

$$\frac{\partial}{\partial \boldsymbol{\vartheta}} E(\boldsymbol{Y}) = \sum_{i,j} \sum_{l} \partial E_{ijl} \frac{\partial Y_{ijl}}{\partial \boldsymbol{\vartheta}}$$

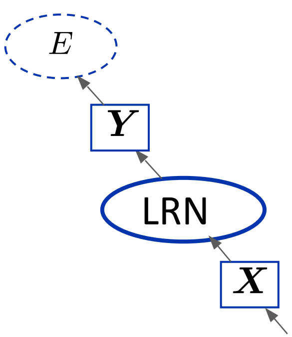
where $\partial E_{ijl} := \frac{\partial E(\boldsymbol{Y})}{\partial Y_{ijl}}$

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

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

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

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



- **Gradient of *Local Response Normalization***

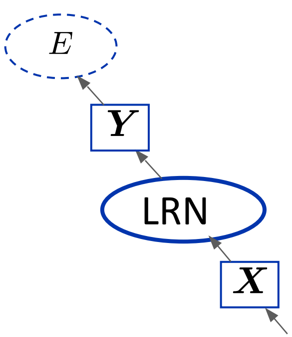
$$Y_{ijl} := \frac{X_{ijl}}{\sum_{k=1}^{h} X_{ijk}}$$

i.e. plain, cross-map normalization

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

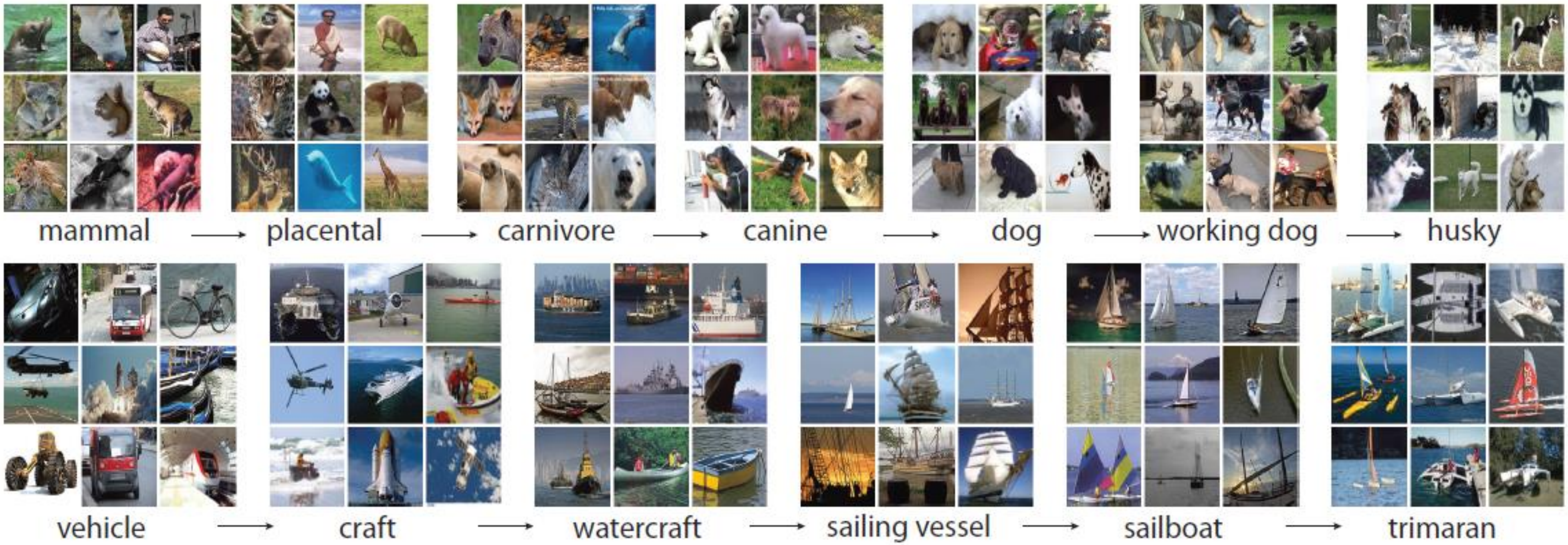
This is inconvenient: the same X components appear multiple times - let's re-factor ….

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



ImageNet Challenge

- The ImageNet Large Scale Visual Recognition Challenge



1,461,406 full resolution images

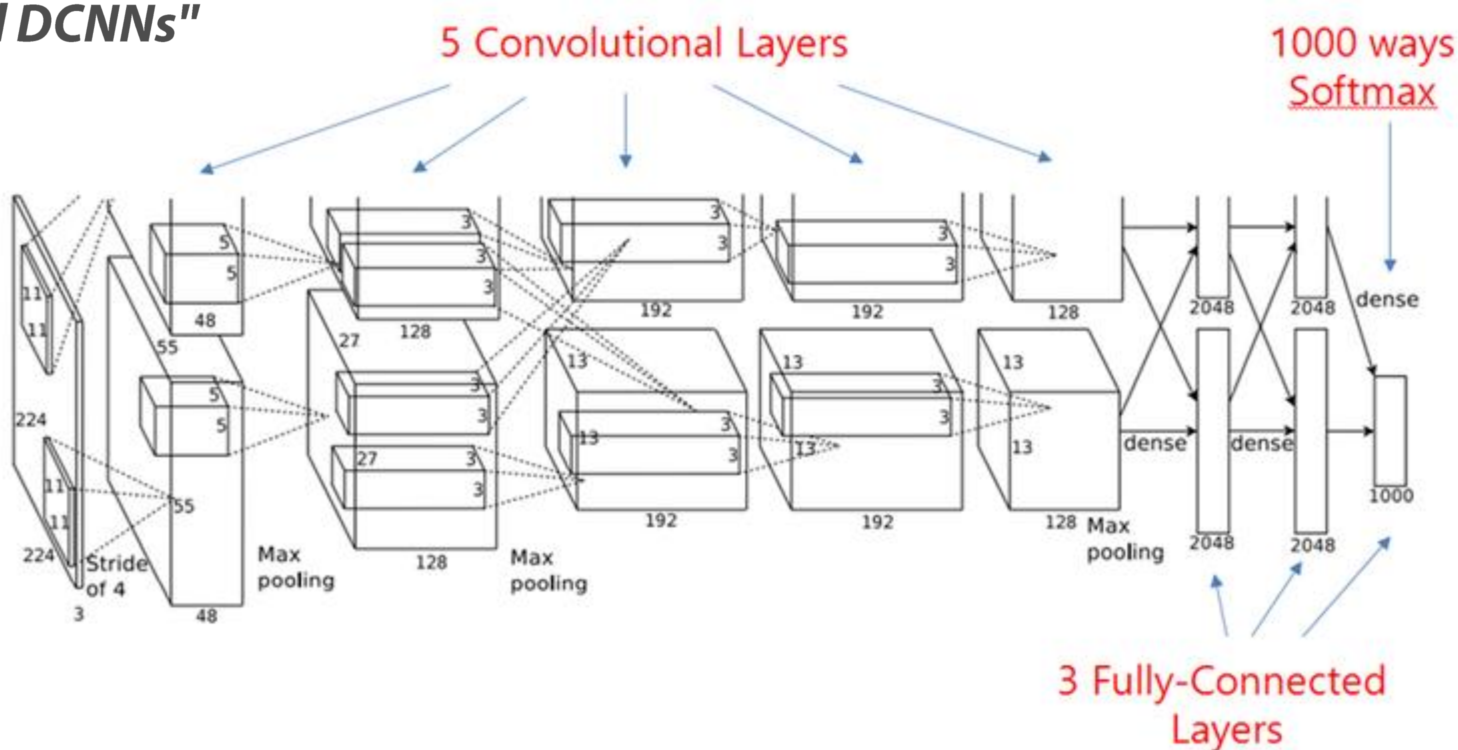
Complex and multiple textual annotation,
hierarchy of 1000 object classes along several dimensions

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

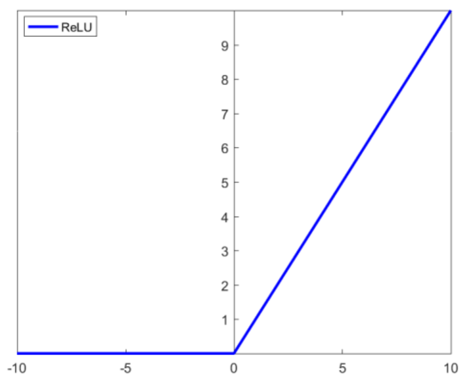
[image from https://world4jason.gitbooks.io/research-log/content/deepLearning/CNN/Model%20&%20ImgNet/alexnet/alexnet.html]

- **AlexNet**

  Why ReLU and not another non-linearity?

  Because it is much faster to train.

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



ReLU

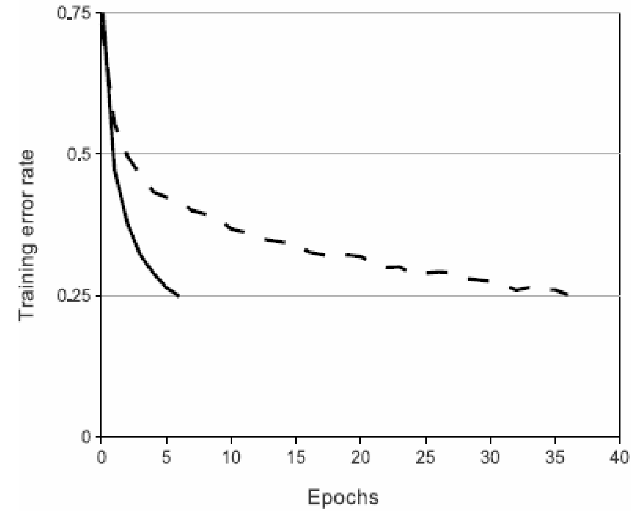


Figure 1: A four-layer convolutional neural network with ReLUs (**solid line**) reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons (**dashed line**). The learning rates for each network were chosen independently to make training as fast as possible. 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.

Image from [Krizhevsky, Sutskever & Hinton, 2012]