

Topics:

- Convolutional Neural Networks

**CS 4644-DL / 7643-A**  
**ZSOLT KIRA**

- **Assignment 2**

- Implement convolutional neural networks

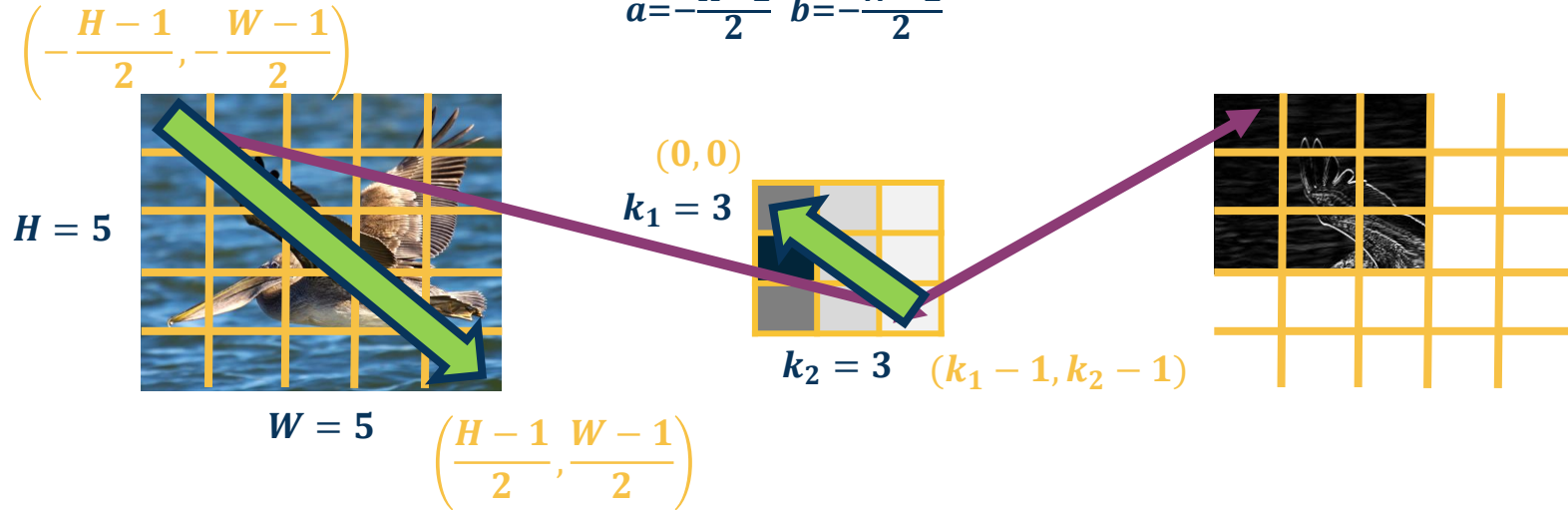
- Resources (in addition to lectures):

- [DL book: Convolutional Networks](#)
- CNN notes [https://www.cc.gatech.edu/classes/AY2022/cs7643\\_spring/assets/L10\\_cnns\\_notes.pdf](https://www.cc.gatech.edu/classes/AY2022/cs7643_spring/assets/L10_cnns_notes.pdf)
- Backprop notes  
[https://www.cc.gatech.edu/classes/AY2022/cs7643\\_spring/assets/L10\\_cnns\\_backprop\\_notes.pdf](https://www.cc.gatech.edu/classes/AY2022/cs7643_spring/assets/L10_cnns_backprop_notes.pdf)
- HW2 Tutorial @113, Conv @116, Focal Loss @117
- Slower OMSCS lectures on dropbox: Module 2 Lessons 5-6 (M2L5/M2L6)  
([https://www.dropbox.com/sh/iviro188gq0b4vs/AADdHxX\\_Uy1TkpF\\_yvIzX0nPa?dl=0](https://www.dropbox.com/sh/iviro188gq0b4vs/AADdHxX_Uy1TkpF_yvIzX0nPa?dl=0))

- FB/Meta Office hours Friday 3pm EST!

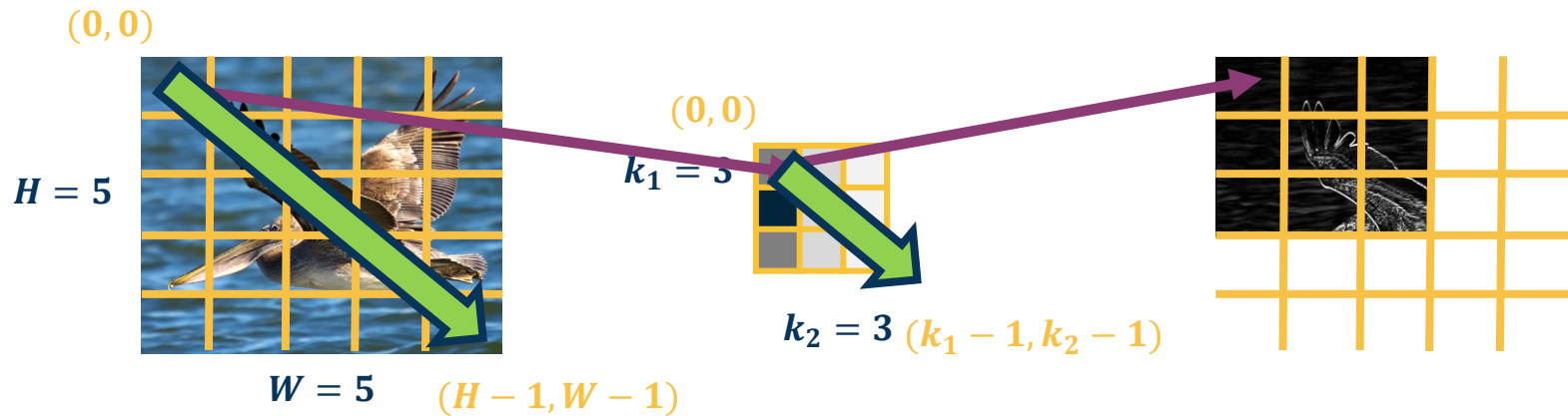
- Pytorch & scalable training
- [Module 2, Lesson 8 \(M2L8\), on dropbox](#)

$$y(r, c) = (x * k)(r, c) = \sum_{a=-\frac{H-1}{2}}^{\frac{H-1}{2}} \sum_{b=-\frac{W-1}{2}}^{\frac{W-1}{2}} x(a, b) k(r - a, c - b)$$



$$y(0, 0) = x(-2, -2)k(2, 2) + x(-2, -1)k(2, 1) + x(-2, 0)k(2, 0) + x(-2, 1)k(2, -1) + x(-2, 2)k(2, -2) + \dots$$

$$y(r, c) = (x * k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r + a, c + b) k(a, b)$$



Since we will be learning these kernels, this change does not matter!

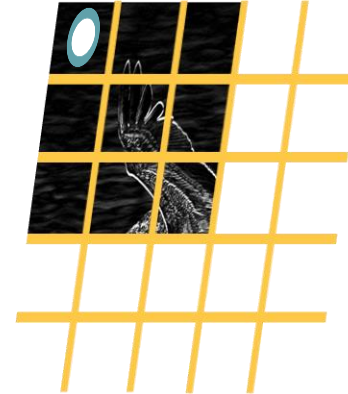
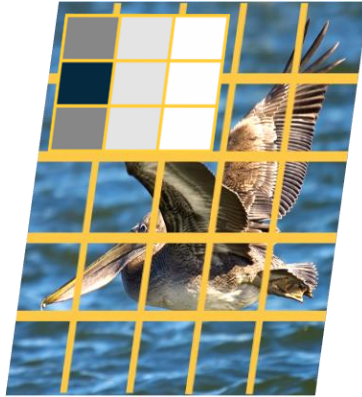
$$x(0:2,0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix}$$

$$K' = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

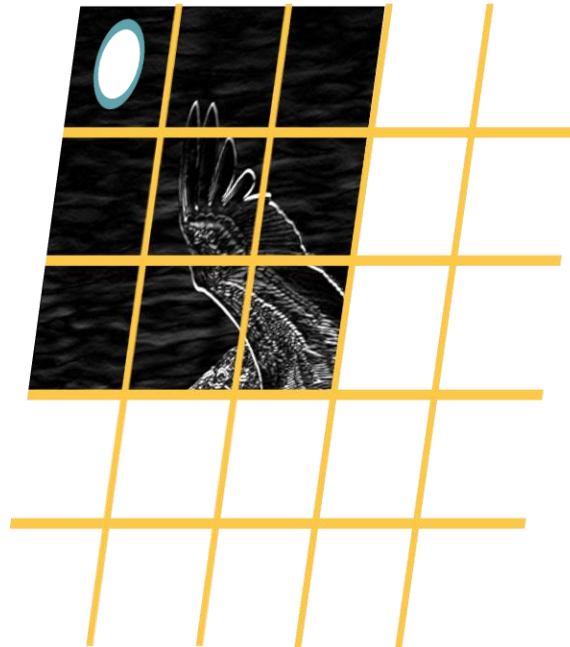
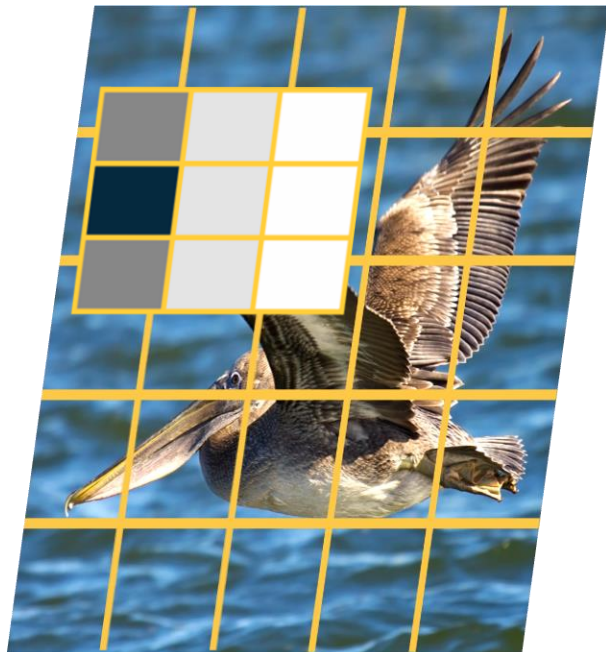


$$x(0:2,0:2) \cdot K' = 65 + \text{bias}$$

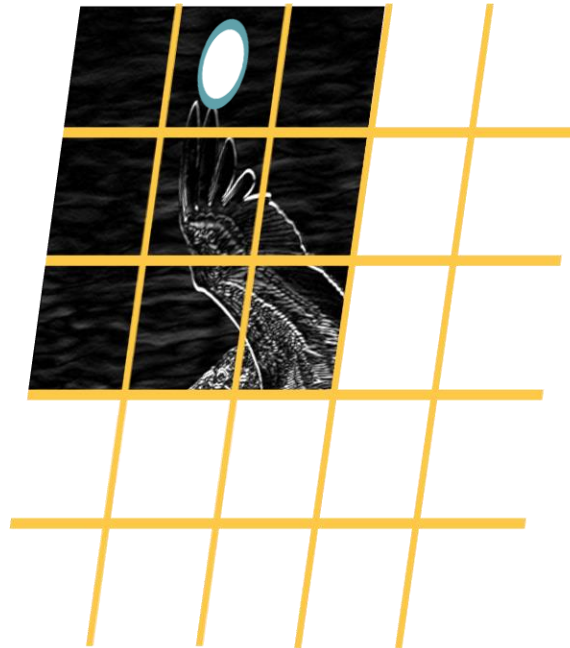
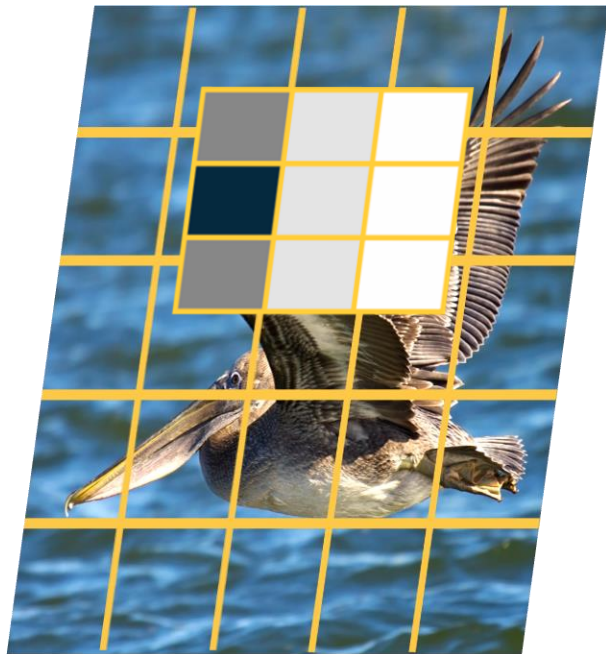
Dot product  
(element-wise multiply and sum)



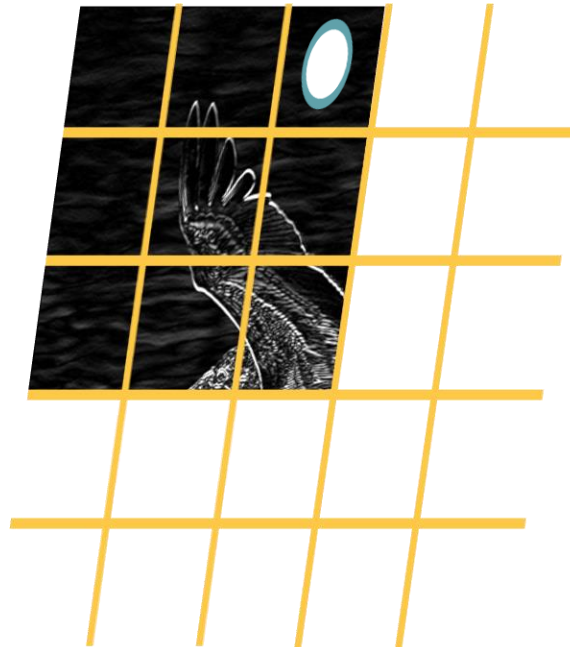
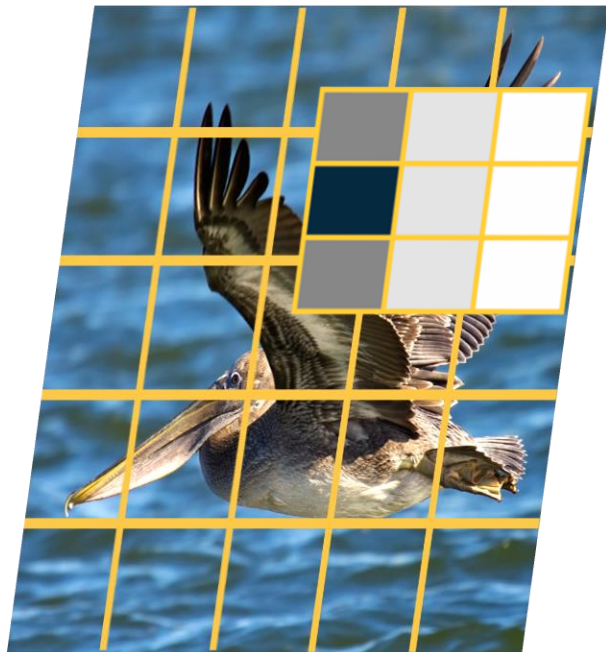
## Cross-Correlation



## Convolution and Cross-Correlation

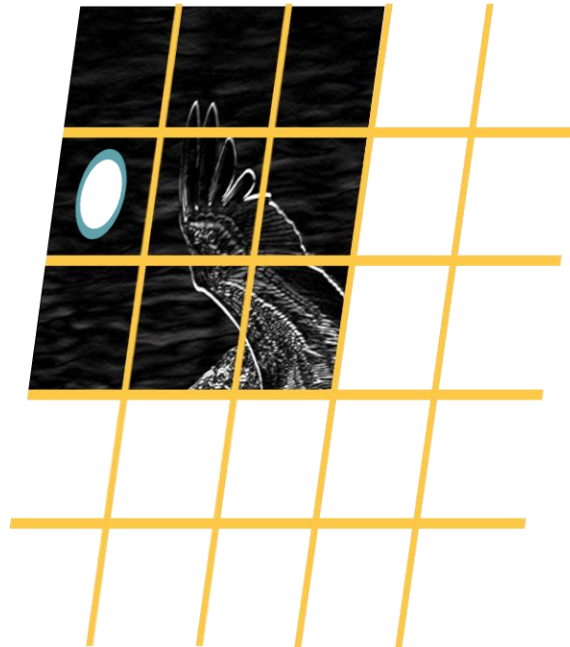
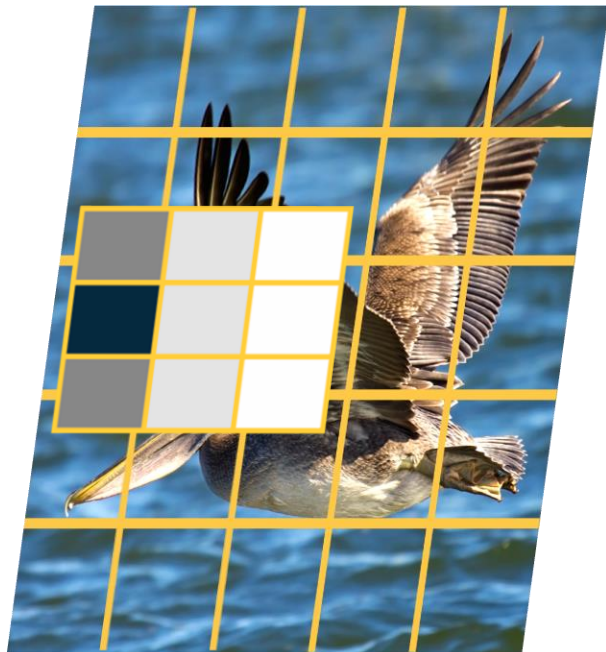


## Convolution and Cross-Correlation

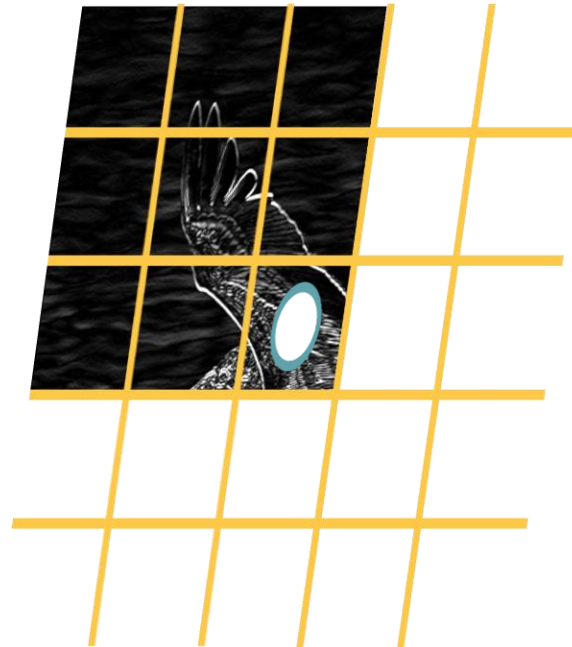
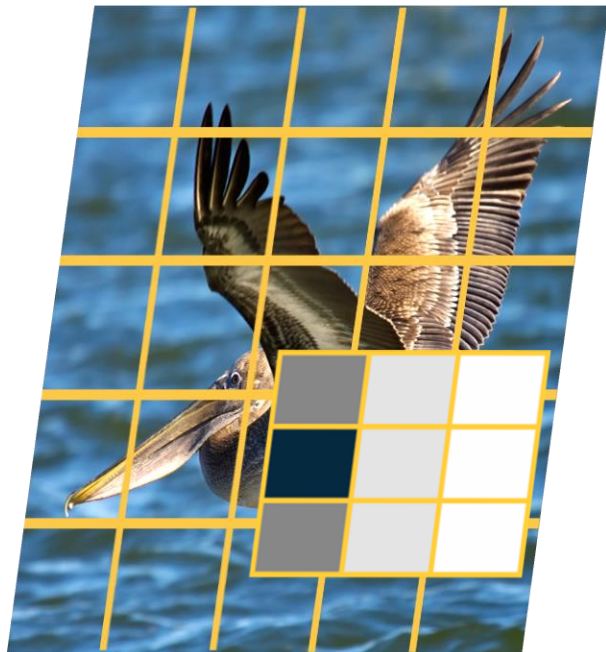


## Convolution and Cross-Correlation





## Convolution and Cross-Correlation



## Convolution and Cross-Correlation

## Why Bother with Convolutions?

Convolutions are just **simple linear operations**

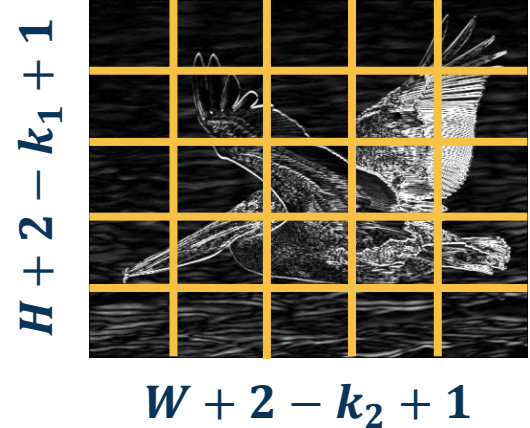
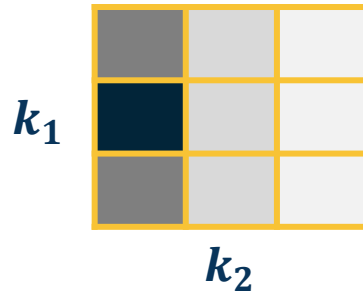
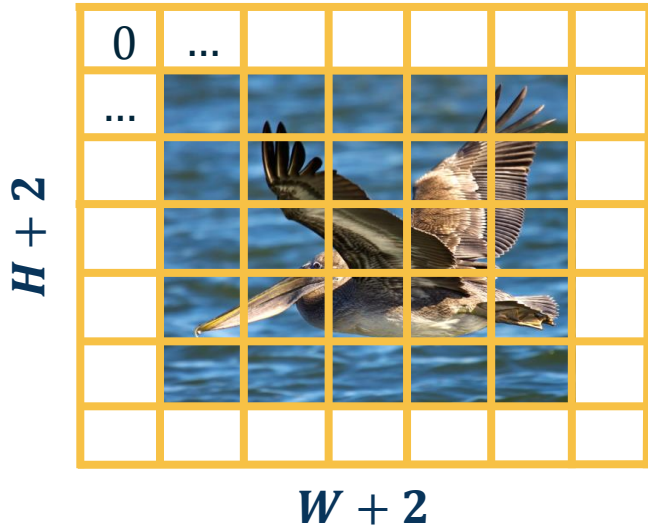
**Why bother** with this and not just say it's a linear layer with small receptive field?

- There is a **duality** between them during backpropagation
- Convolutions have **various mathematical properties** people care about
- This is **historically** how it was inspired



We can **pad the images** to make the output the same size:

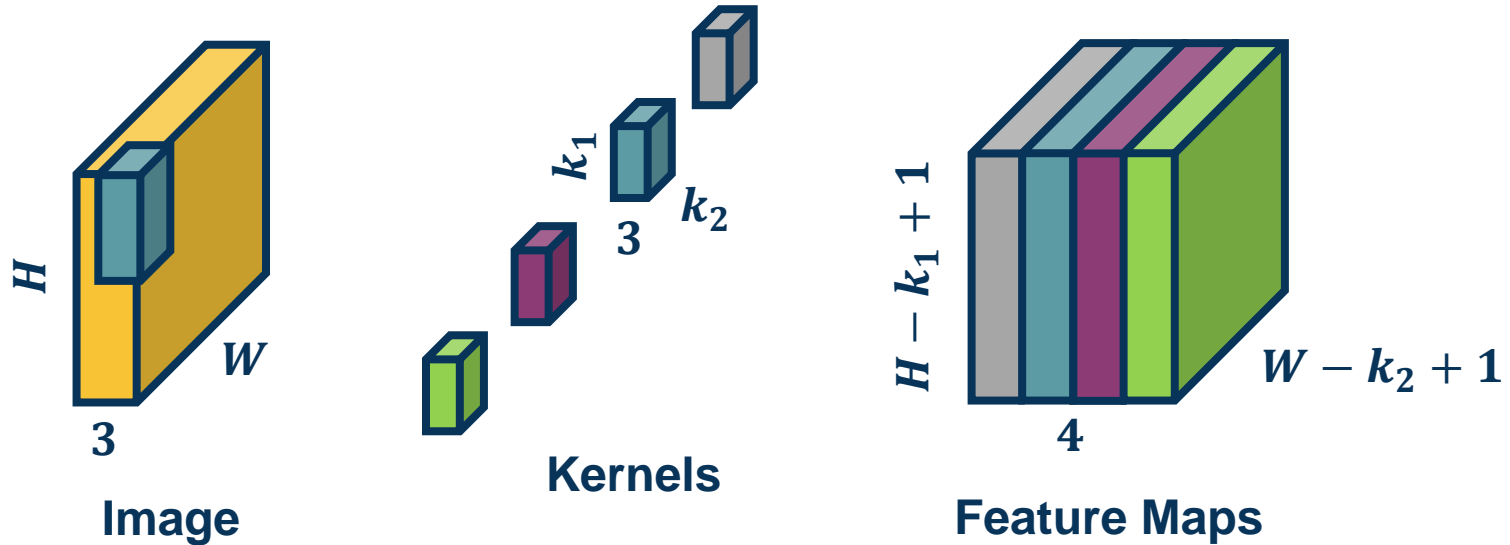
- Zeros, mirrored image, etc.
- Note padding often refers to pixels added to **one size** ( $P = 1$  here)



We can have **multiple kernels per layer**

- ▶ We stack the feature maps together at the output

Number of  
channels in output  
is equal to *number  
of kernels*

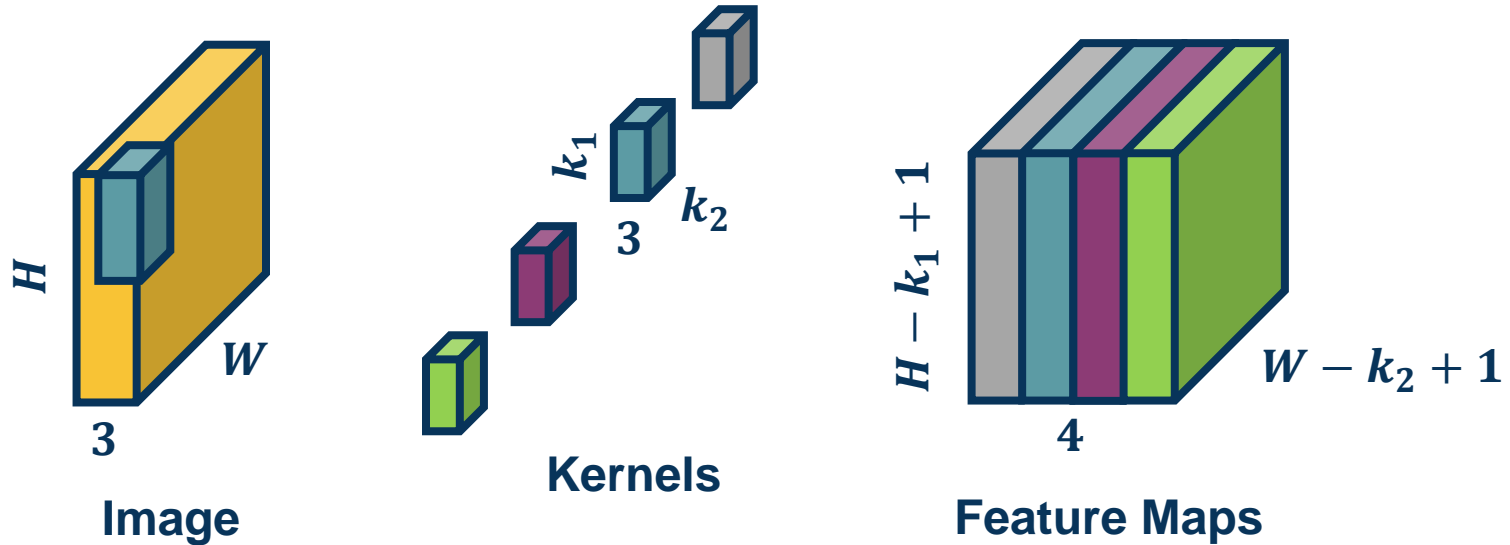


## Multiple Kernels

Number of parameters with  $N$  filters is:  $N * (k_1 * k_2 * 3 + 1)$

Example:

$k_1 = 3, k_2 = 3, N = 4$  *input channels* = 3, then  $(3 * 3 * 3 + 1) * 4 = 112$



Number of Parameters

Need to incorporate all upstream gradients:

$$\left\{ \frac{\partial L}{\partial y(0,0)}, \frac{\partial L}{\partial y(0,1)}, \dots, \frac{\partial L}{\partial y(H,W)} \right\}$$

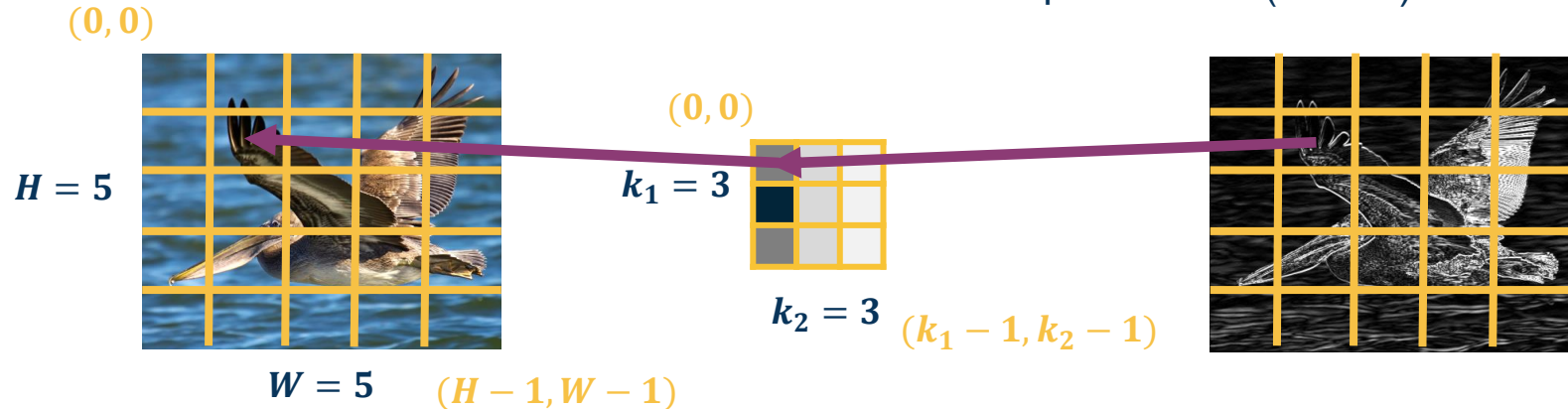
Chain Rule:

$$\frac{\partial L}{\partial k(a,b)} = \sum_{r=0}^{H-1} \sum_{c=0}^{W-1} \frac{\partial L}{\partial y(r,c)} \frac{\partial y(r,c)}{\partial k(a,b)}$$

Sum over  
all output  
pixels

Upstream  
gradient  
(known)

We will  
compute

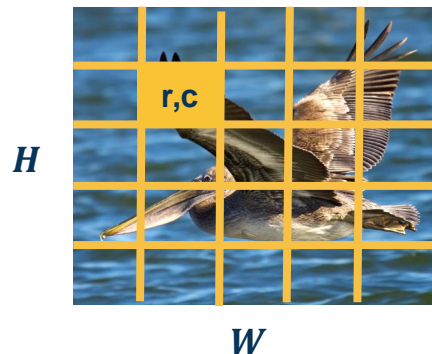
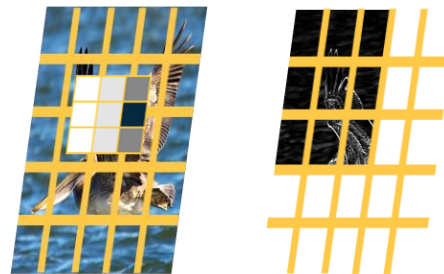


Chain Rule over all Output Pixels

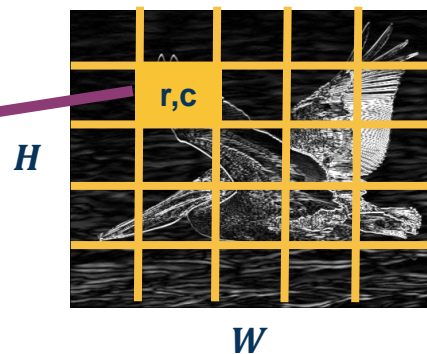
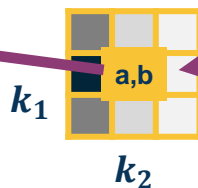
$$\frac{\partial y(r, c)}{\partial k(a, b)} = ?$$

### Reasoning:

- Cross-correlation is just “dot product” of kernel and input patch (weighted sum)
- When at pixel  $y(r, c)$ , kernel is on input  $x$  such that  $k(0, 0)$  is multiplied by  $x(r, c)$
- But we want derivative w.r.t.  $k(a, b)$ 
  - $k(0, 0) * x(r, c)$ ,  $k(1, 1) * x(r + 1, c + 1)$ ,  $k(2, 2) * x(r + 2, c + 2) \Rightarrow$  in general  $k(a, b) * x(r + a, c + b)$
  - Just like before in fully connected layer, partial derivative w.r.t.  $k(a, b)$  only has this term (other  $x$  terms go away because not multiplied by  $k(a, b)$ ).



?



Chain Rule over all Output Pixels

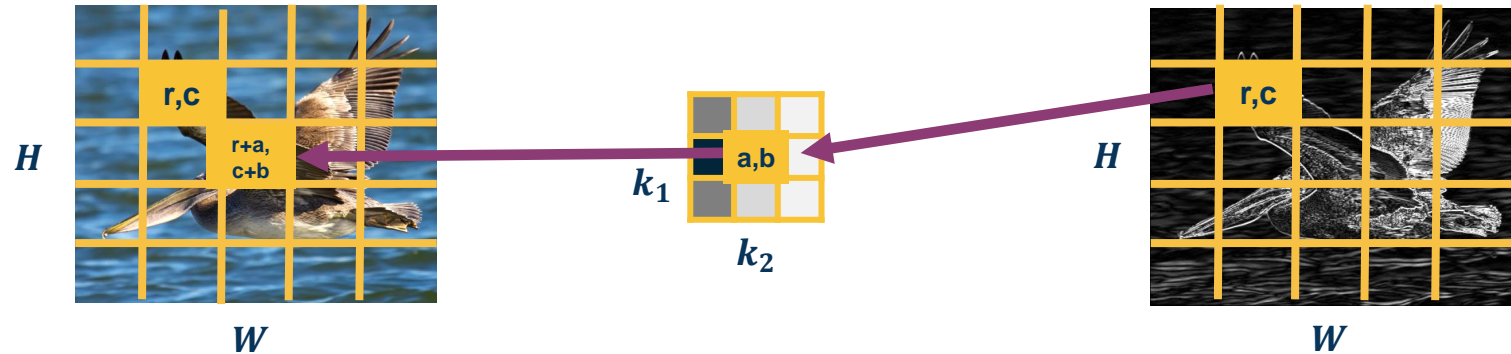


$$\frac{\partial y(r, c)}{\partial k(a, b)} = x(r + a, c + b)$$

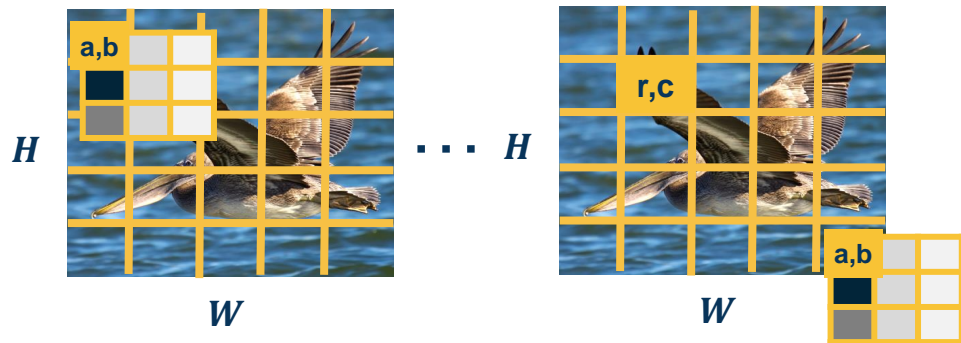
$$\frac{\partial L}{\partial k(a, b)} = \sum_{r=0}^{H-1} \sum_{c=0}^{W-1} \frac{\partial L}{\partial y(r, c)} x(r + a, c + b)$$

Does this look familiar?

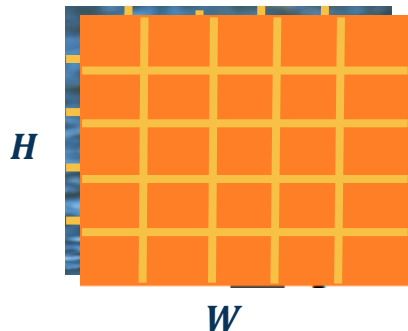
Cross-correlation  
between upstream  
gradient and input!  
(until  $k_1 \times k_2$  output)



## Forward Pass



## Backward Pass $k(0,0)$



## Backward Pass $k(2,2)$



Does this look familiar?

Cross-correlation  
between upstream  
gradient and input!  
(until  $k_1 \times k_2$  output)



Forward and Backward Duality

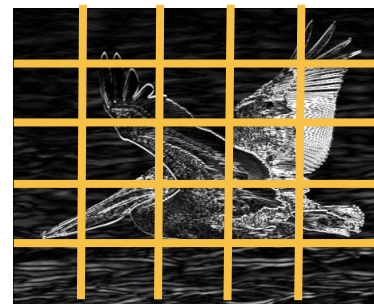
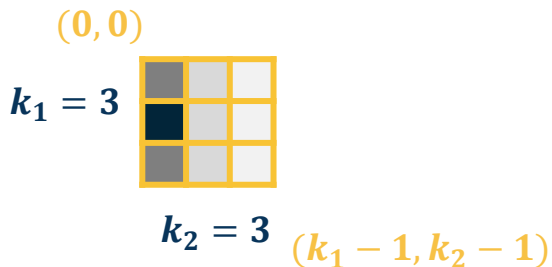
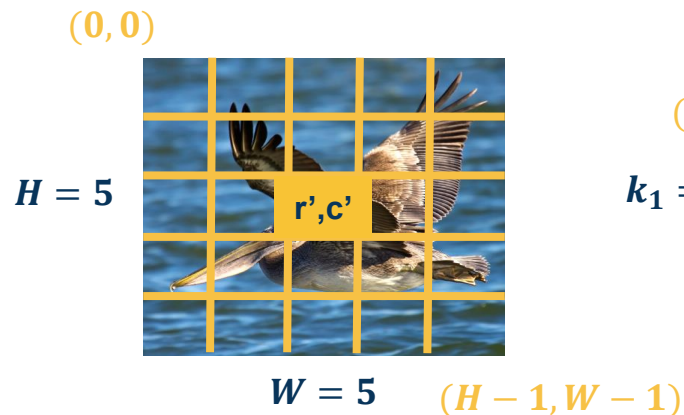
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x}$$

Gradient for input (to pass to prior layer)

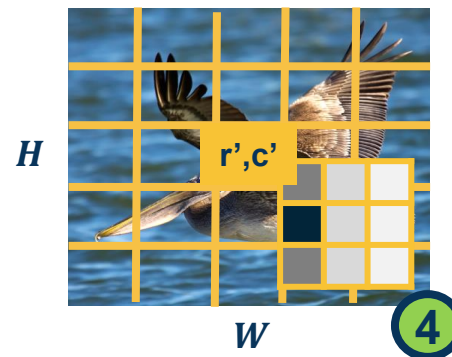
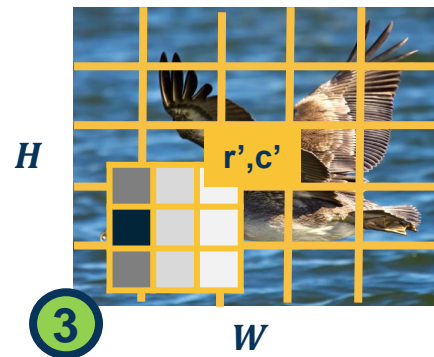
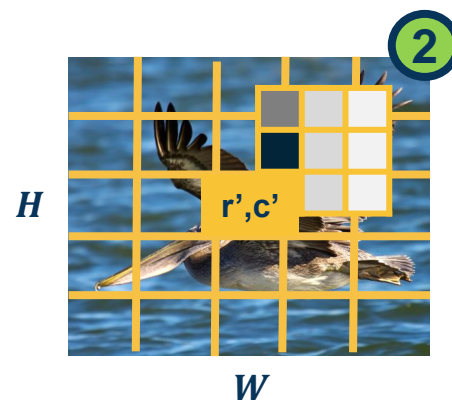
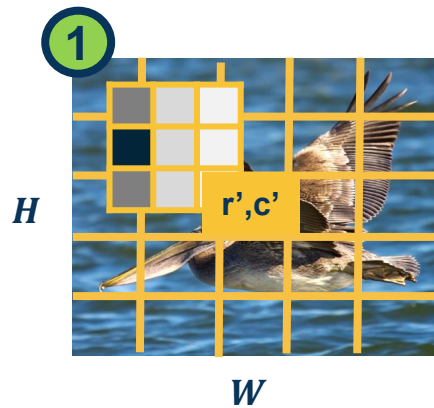
Calculate one pixel at a time  $\frac{\partial L}{\partial x(r', c')}$

What does this input pixel affect at the output?

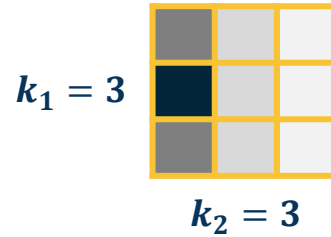
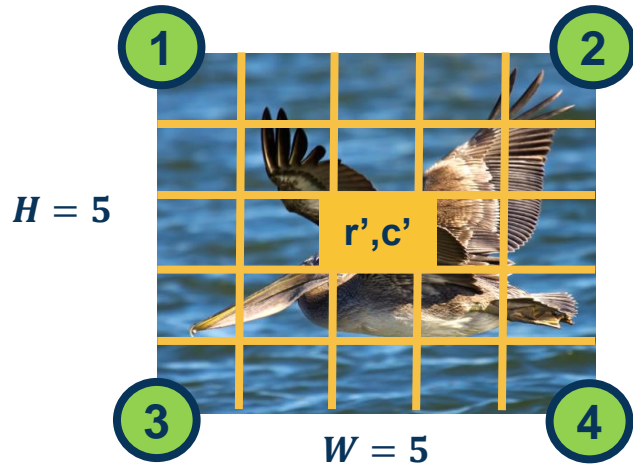
Neighborhood around it  
(where part of the kernel touches it)



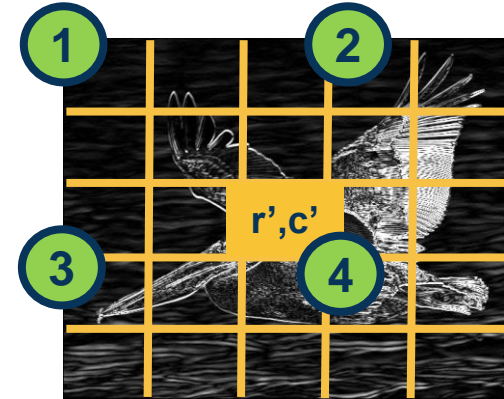
What an Input Pixel Affects at Output



Extents of Kernel Touching the Pixel



$$(r' - k_1 + 1, \\ c' - k_2 + 1)$$



This is where the corresponding locations are for the **output**

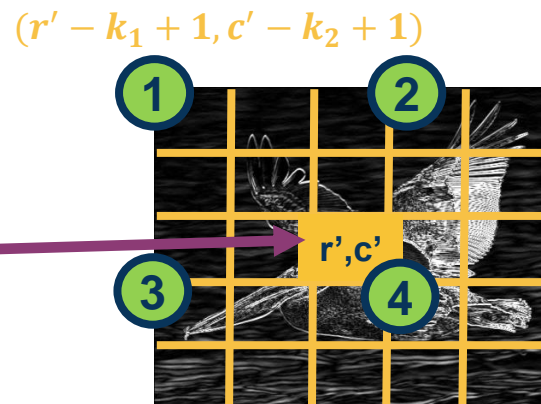
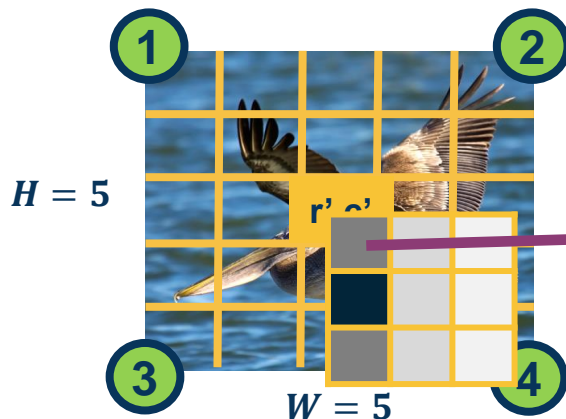
## Extents at the Output

Chain rule for affected pixels (sum gradients):

$$\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}$$

$$\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(?, ?)} \frac{\partial y(?, ?)}{\partial x(r', c')}$$

$$\begin{aligned} x(r', c') * k(0, 0) &\Rightarrow y(r', c') \\ x(r', c') * k(1, 1) &\Rightarrow ? \end{aligned}$$



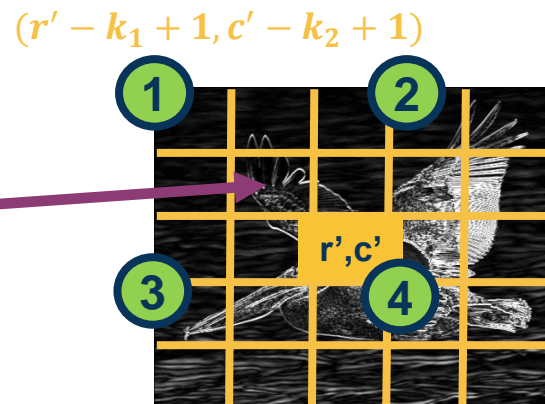
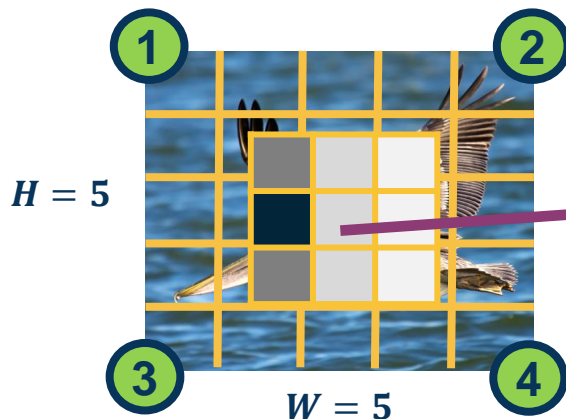
Summing Gradient Contributions

Chain rule for affected pixels (sum gradients):

$$\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}$$

$$\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(?, ?)} \frac{\partial y(?, ?)}{\partial x(r', c')}$$

$$\begin{aligned} x(r', c') * k(0, 0) &\Rightarrow y(r', c') \\ x(r', c') * k(1, 1) &\Rightarrow y(r' - 1, c' - 1) \\ \dots \\ x(r', c') * k(a, b) &\Rightarrow y(r' - a, c' - b) \end{aligned}$$



Summing Gradient Contributions

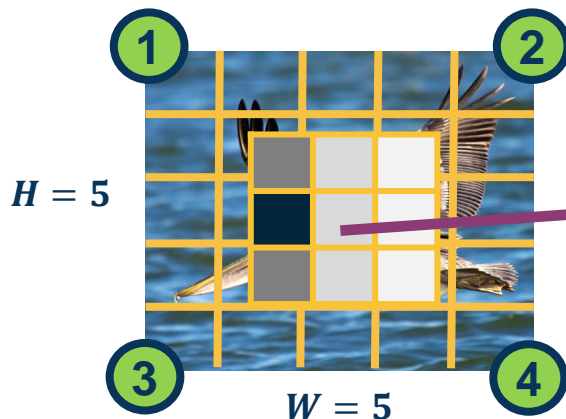


Chain rule for affected pixels (sum gradients):

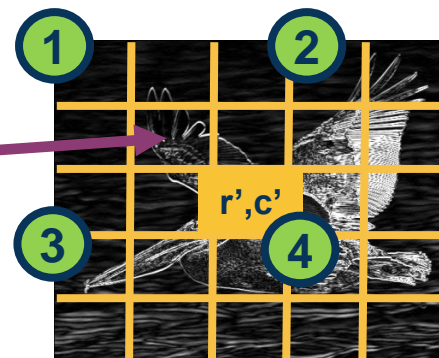
$$\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}$$

$$\frac{\partial L}{\partial x(r', c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} \frac{\partial y(r' - a, c' - b)}{\partial x(r', c')}$$

Let's derive it  
analytically this time (as  
opposed to visually)



$(r' - k_1 + 1, c' - k_2 + 1)$



Summing Gradient Contributions



Definition of cross-correlation (use  $a', b'$  to distinguish from prior variables):

$$y(\mathbf{r}', \mathbf{c}') = (x * \mathbf{k})(\mathbf{r}', \mathbf{c}') = \sum_{a'=0}^{k_1-1} \sum_{b'=0}^{k_2-1} x(\mathbf{r}' + \mathbf{a}', \mathbf{c}' + \mathbf{b}') \mathbf{k}(\mathbf{a}', \mathbf{b}')$$

Plug in what we actually wanted :

$$y(\mathbf{r}' - \mathbf{a}, \mathbf{c}' - \mathbf{b}) = (x * \mathbf{k})(\mathbf{r}', \mathbf{c}') = \sum_{a'=0}^{k_1-1} \sum_{b'=0}^{k_2-1} x(\mathbf{r}' - \mathbf{a} + \mathbf{a}', \mathbf{c}' - \mathbf{b} + \mathbf{b}') \mathbf{k}(\mathbf{a}', \mathbf{b}')$$

What is  $\frac{\partial y(\mathbf{r}' - \mathbf{a}, \mathbf{c}' - \mathbf{b})}{\partial x(\mathbf{r}', \mathbf{c}')} = \mathbf{k}(\mathbf{a}, \mathbf{b})$  (we want term with  $x(\mathbf{r}', \mathbf{c}')$  in it;  
this happens when  $\mathbf{a} = \mathbf{a}'$  and  $\mathbf{b} = \mathbf{b}'$ )

Plugging in to earlier equation:

$$\begin{aligned}\frac{\partial L}{\partial x(r', c')} &= \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} \frac{\partial y(r' - a, c' - b)}{\partial x(r', c')} \\ &= \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(r' - a, c' - b)} k(a, b)\end{aligned}$$

Again, all operations can be implemented via matrix multiplications (same as FC layer)!

Does this look familiar?

Convolution between upstream gradient and kernel!

(can implement by flipping kernel and cross-correlation)

Backwards is Convolution

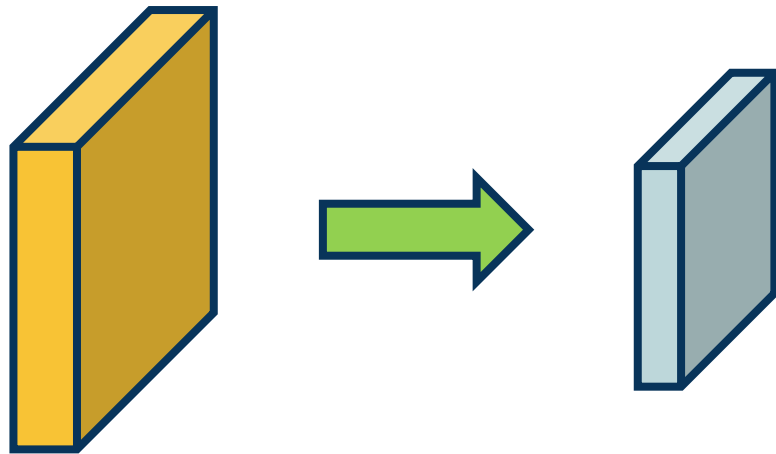
- Convolutions are mathematical descriptions of striding linear operation
- In practice, we implement **cross-correlation neural networks!** (still called convolutional neural networks due to history)
  - Can connect to convolutions via duality (flipping kernel)
  - Convolution formulation has mathematical properties explored in ECE
- Duality for forwards and backwards:
  - **Forward:** Cross-correlation
  - **Backwards w.r.t.  $K$ :** Cross-correlation b/w upstream gradient and input
  - **Backwards w.r.t.  $X$ :** Convolution b/w upstream gradient and kernel
    - In practice implement via cross-correlation and flipped kernel
- All operations still implemented via **efficient linear algebra** (e.g. matrix-matrix multiplication)

# Pooling Layers

➤ **Dimensionality reduction** is an important aspect of machine learning

➤ Can we make a layer to **explicitly down-sample** image or feature maps?

➤ **Yes!** We call one class of these operations **pooling** operations



#### Parameters

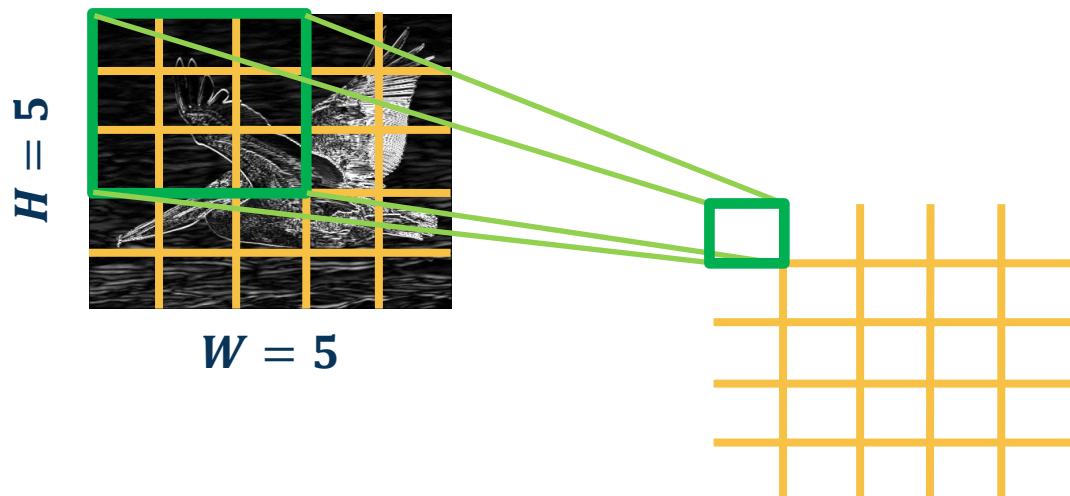
- **kernel\_size** – the size of the window to take a max over
- **stride** – the stride of the window. Default value is `kernel_size`
- **padding** – implicit zero padding to be added on both sides

From: <https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html#torch.nn.MaxPool2d>

## Example: Max pooling

- Stride window across image but perform per-patch **max operation**

$$X(0:2, 0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix} \Rightarrow \max(0:2, 0:2) = 200$$



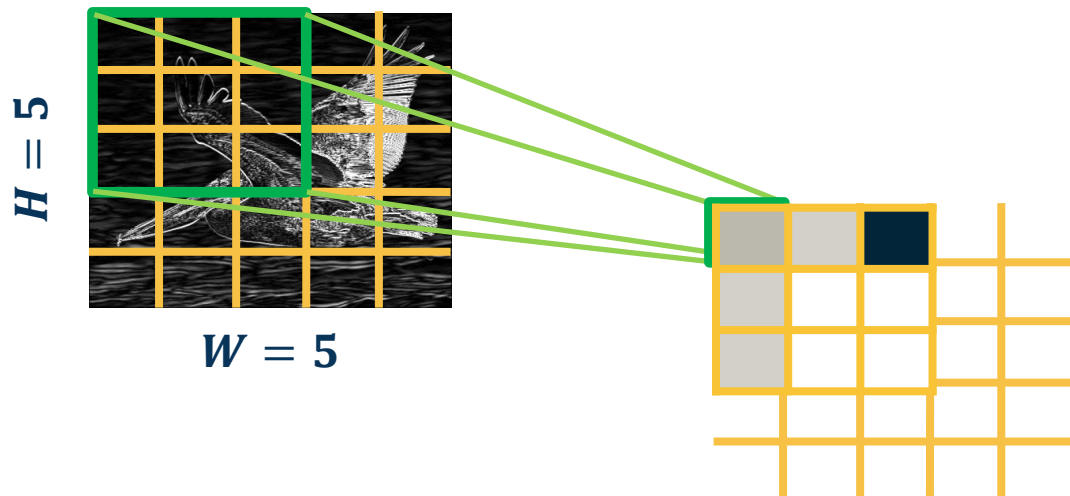
How many learned parameters does this layer have?

None!

**Not restricted to max;** can use any differentiable function

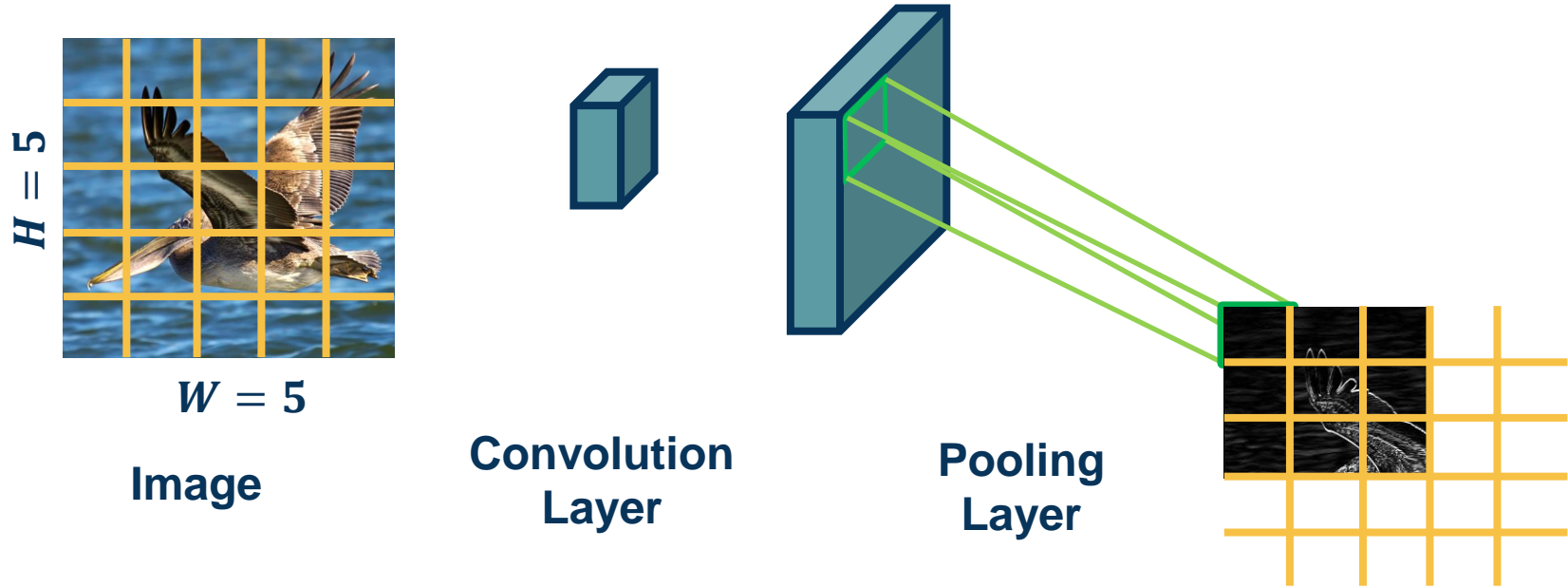
✪ Not very common in practice

$$X(0:2, 0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix} \Rightarrow \text{average}(0:2, 0:2) = \frac{1}{N} \sum_i \sum_j x(i, j) = 90$$



**Max Pooling**

Since the **output** of convolution and pooling layers are **(multi-channel) images**, we can sequence them just as any other layer

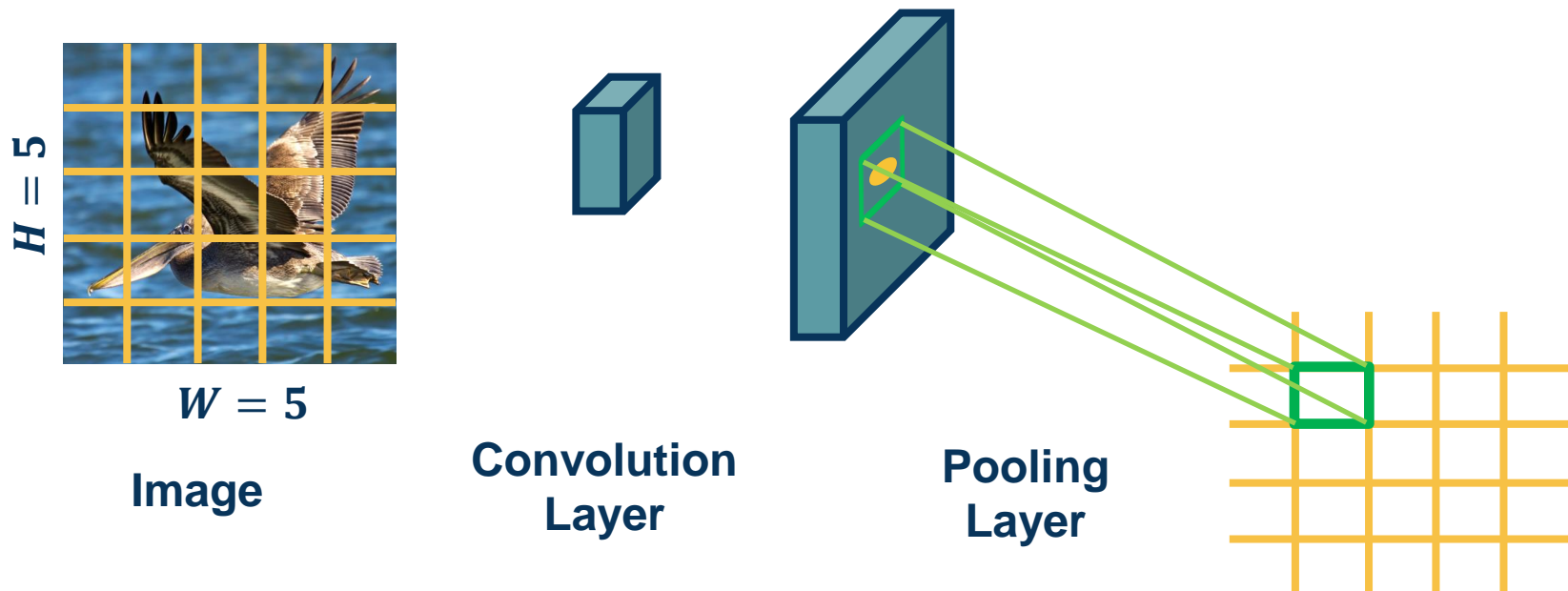


## Combining Convolution & Pooling Layers



This combination adds some **invariance** to translation of the features

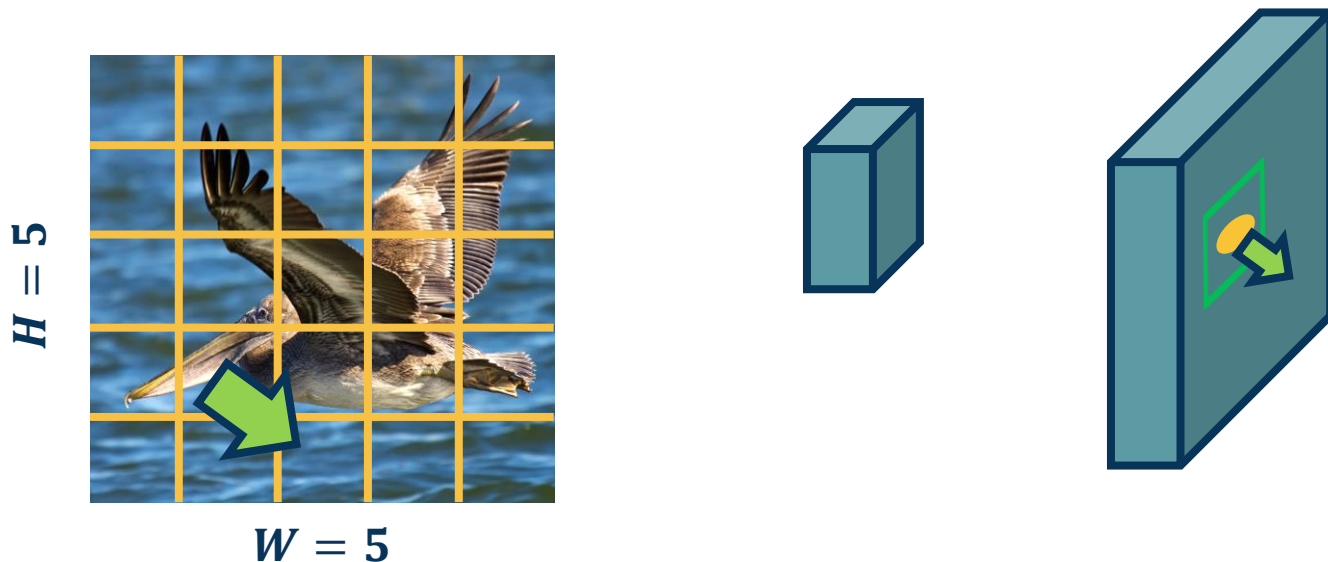
- If feature (such as beak) translated a little bit, output values still **remain the same**



**Invariance**

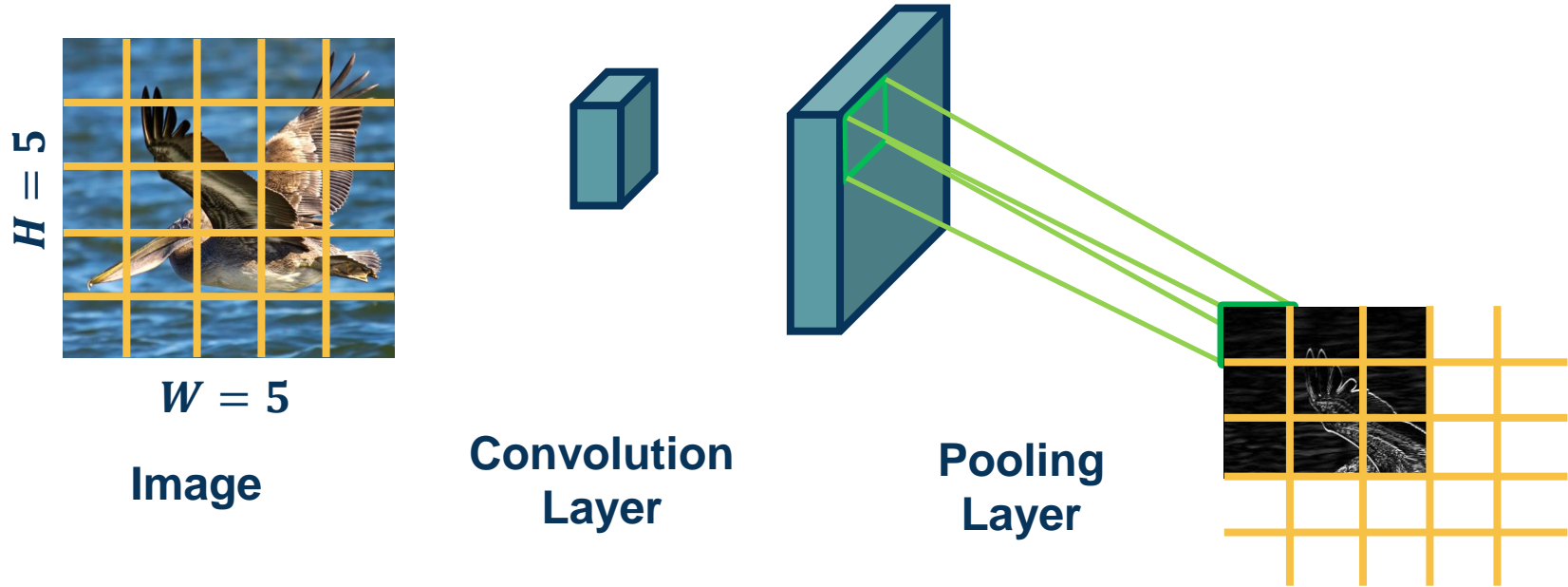
Convolution by itself has the property of **equivariance**

- If feature (such as beak) translated a little bit, output values **move by the same translation**



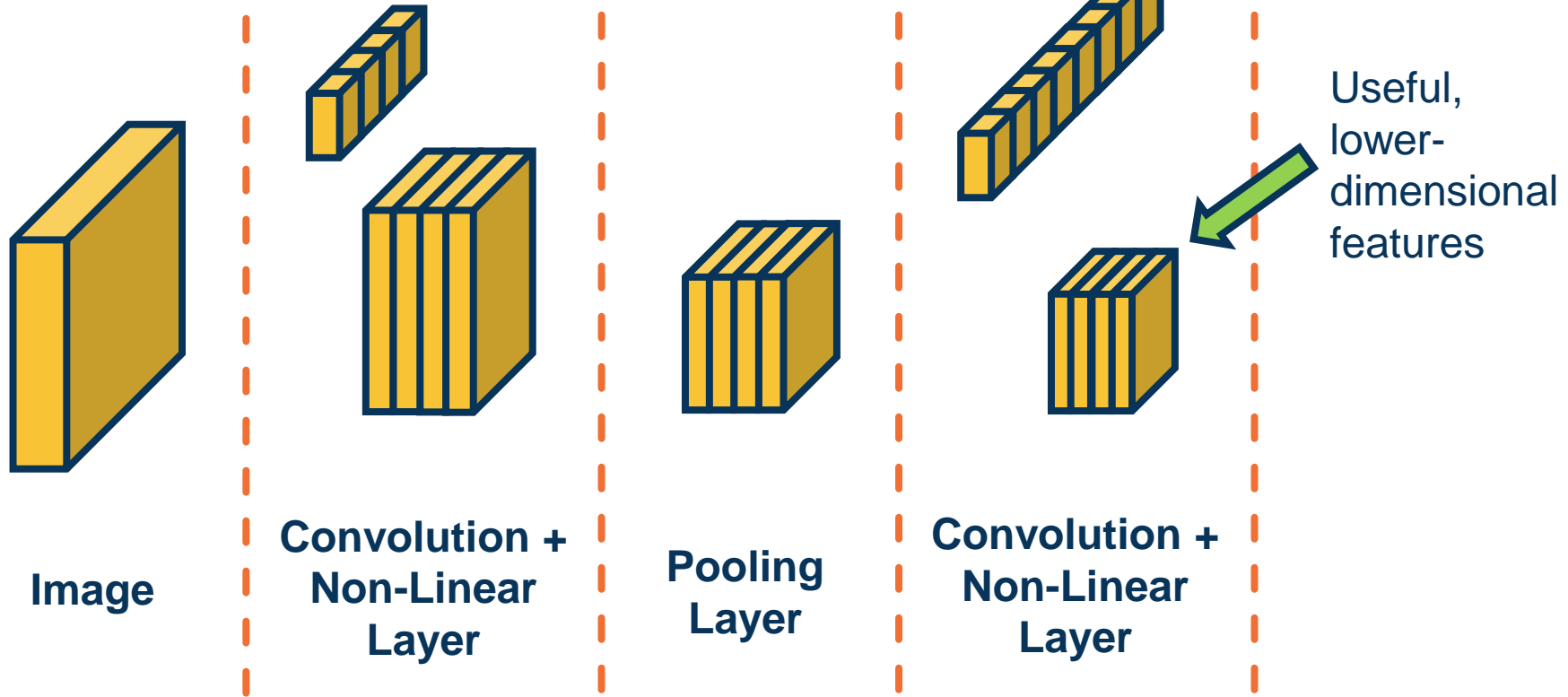
# **Simple Convolutional Neural Networks**

Since the **output** of convolution and pooling layers are **(multi-channel) images**, we can sequence them just as any other layer

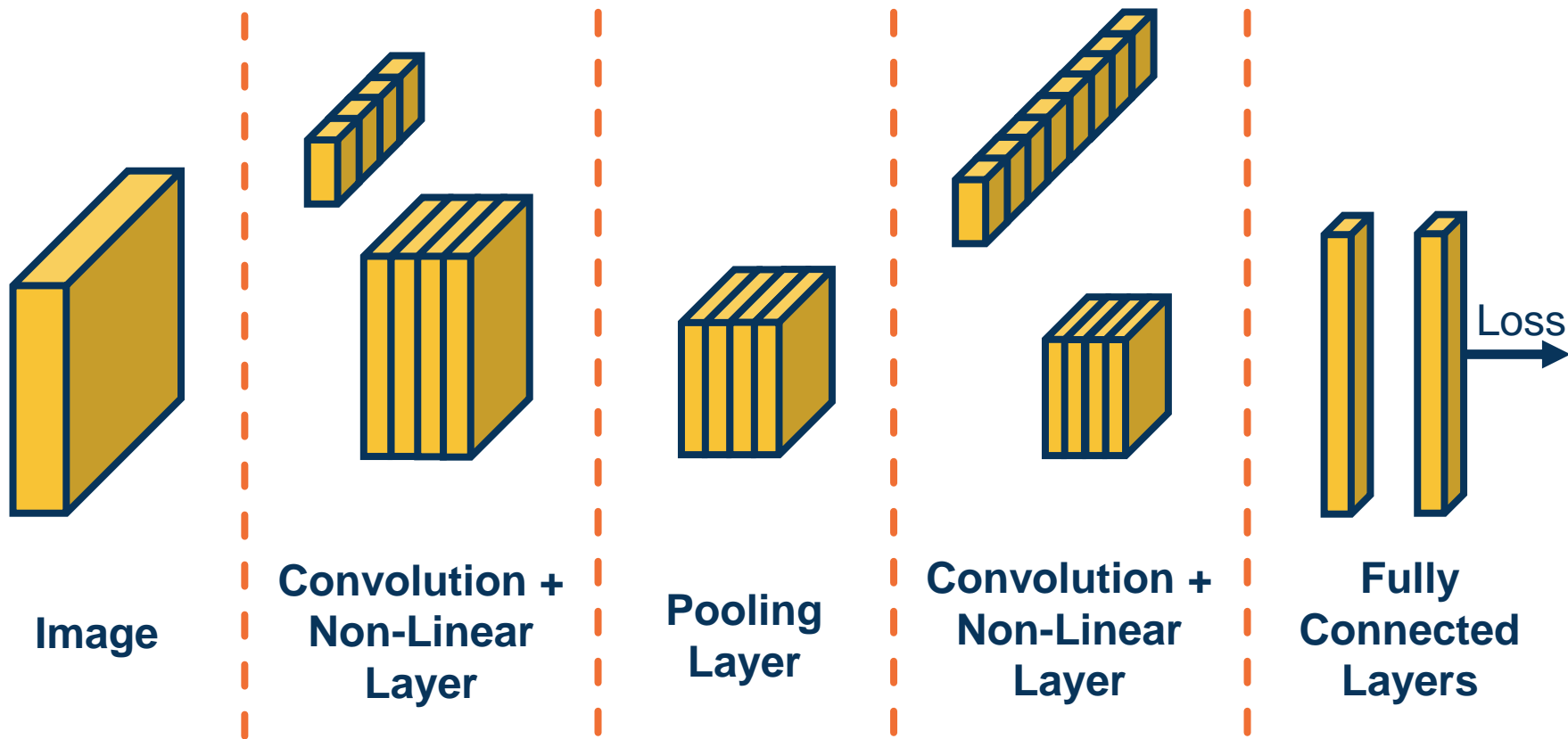


## Combining Convolution & Pooling Layers

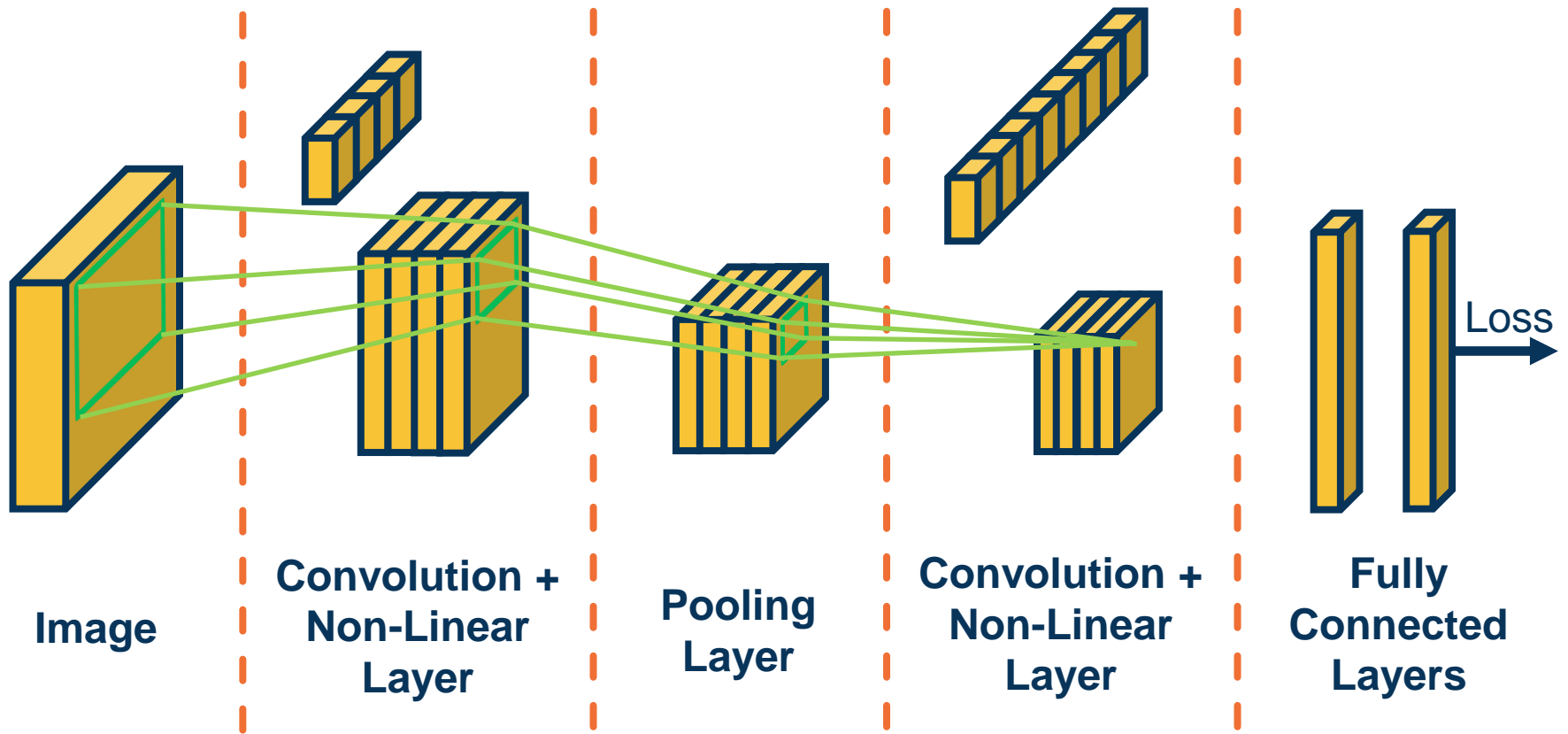
# Convolutional Neural Networks (CNNs)



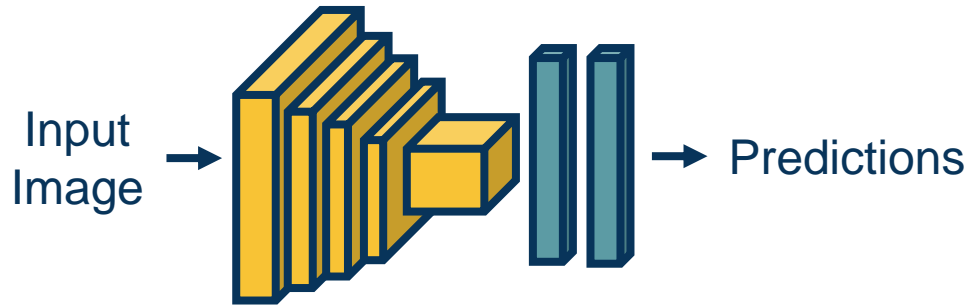
**Alternating Convolution and Pooling**



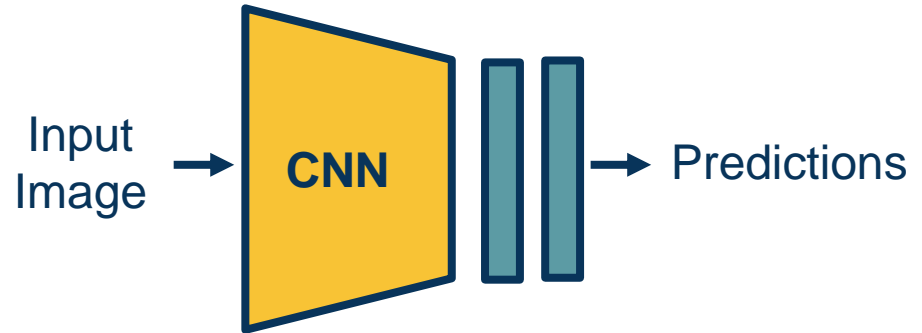
**Adding a Fully Connected Layer**



**Receptive Fields**



## Convolutional Neural Networks



Typical Depiction of CNNs



These architectures have existed **since 1980s**

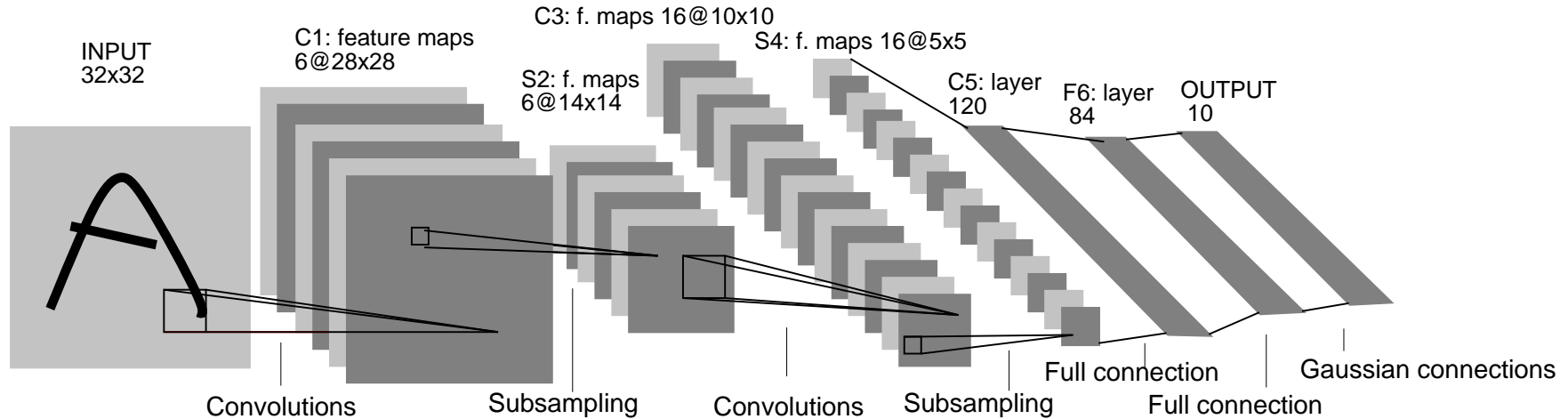


Image Credit: Yann LeCun, Kevin Murphy

## LeNet Architecture

# Handwriting Recognition

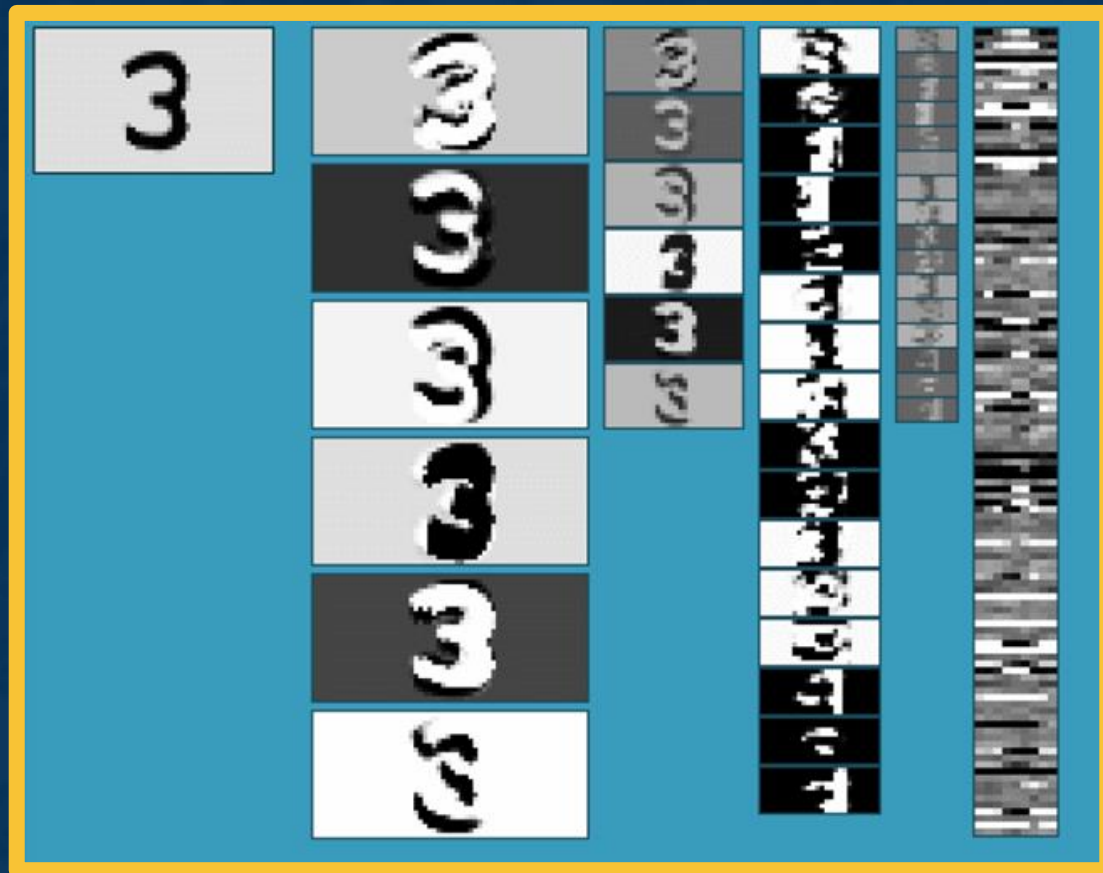


Image Credit:  
Yann LeCun

# Translation Equivariance (Conv Layers) & Invariance (Output)



Image Credit:  
Yann LeCun

## (Some) Rotation Invariance

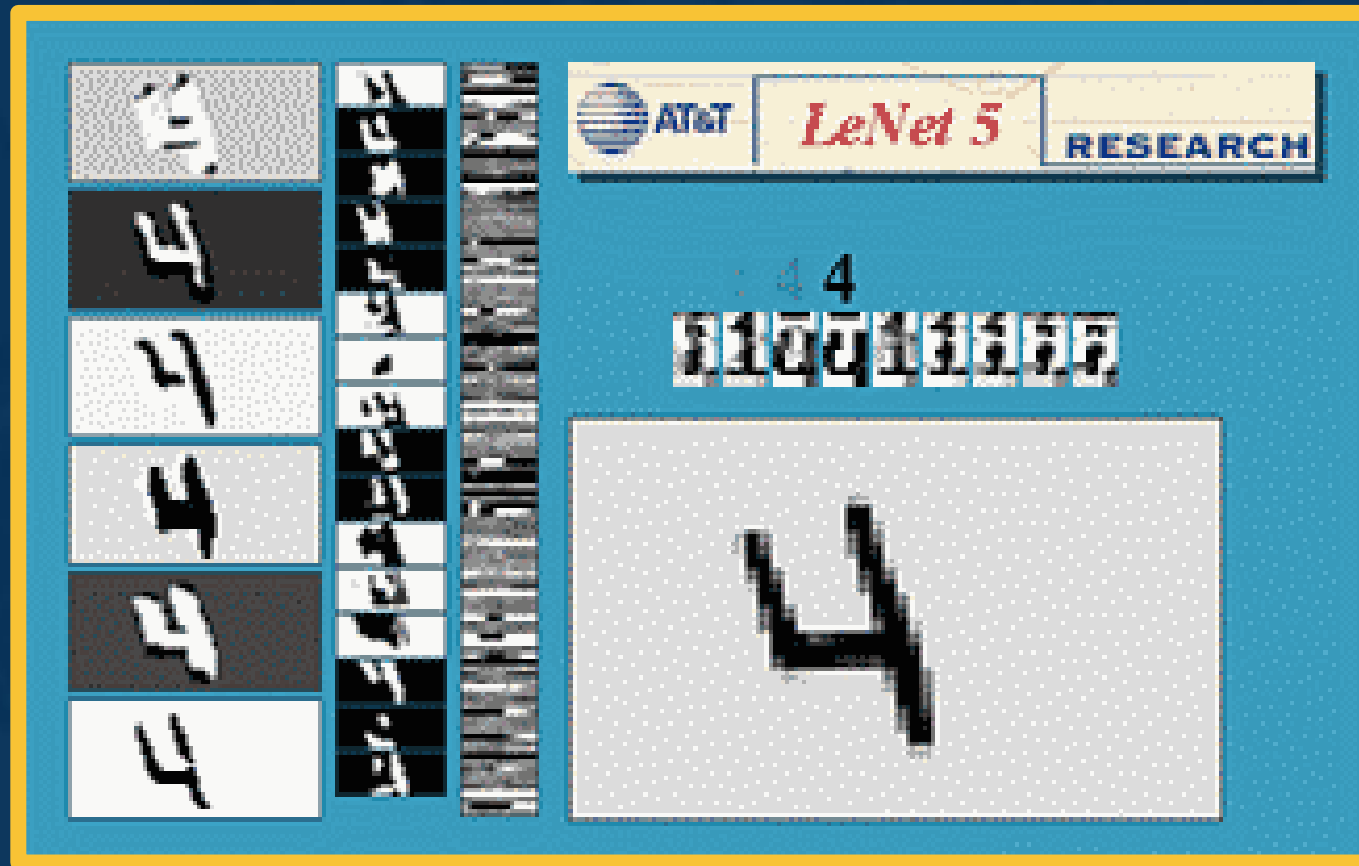


Image Credit:  
Yann LeCun

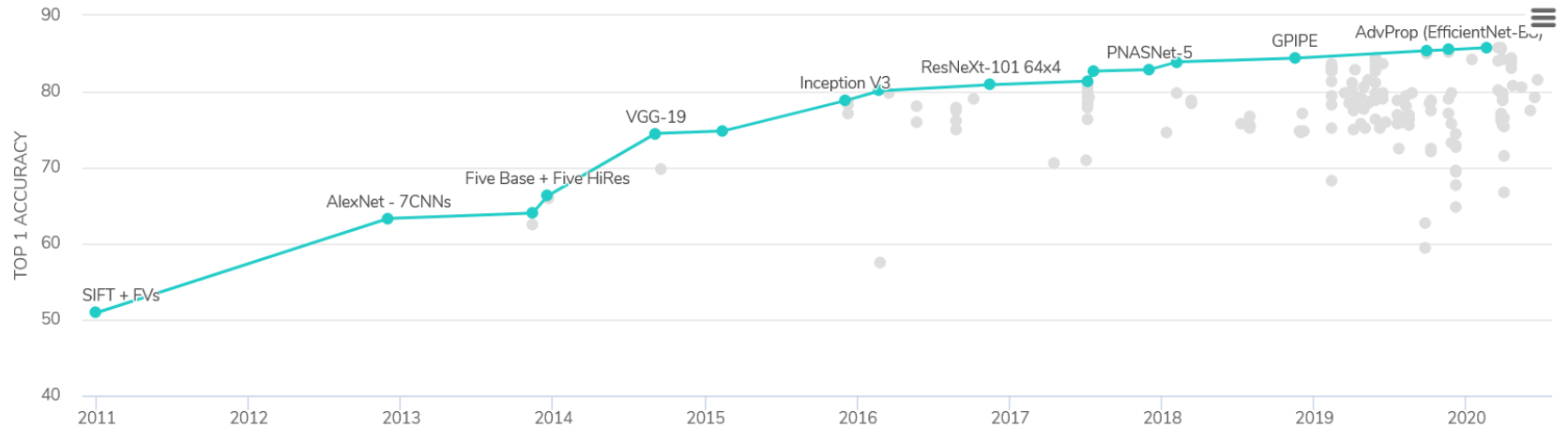
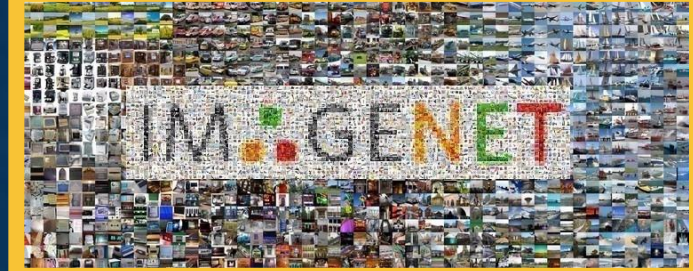
## (Some) Scale Invariance



Image Credit:  
Yann LeCun

# **Advanced Convolutional Networks**

# The Importance of Benchmarks



From: <https://paperswithcode.com>

The diagram illustrates a deep convolutional neural network (CNN) architecture for handwritten digit recognition. It starts with an input layer of size 224x224x3. The first layer is a convolutional layer with a kernel size of 11x11 and a stride of 4, resulting in two feature maps of size 55x55x48. The second layer is a max pooling layer, resulting in two feature maps of size 27x27x128. The third layer is a convolutional layer with a kernel size of 3x3, resulting in two feature maps of size 13x13x192. The fourth layer is a max pooling layer, resulting in two feature maps of size 13x13x128. The fifth layer is a convolutional layer with a kernel size of 3x3, resulting in two feature maps of size 13x13x128. The sixth layer is a max pooling layer, resulting in two feature maps of size 13x13x128. The seventh layer is a dense layer of size 2048. The eighth layer is a dense layer of size 2048. The final layer is a dense layer of size 1000, representing the output classes.





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

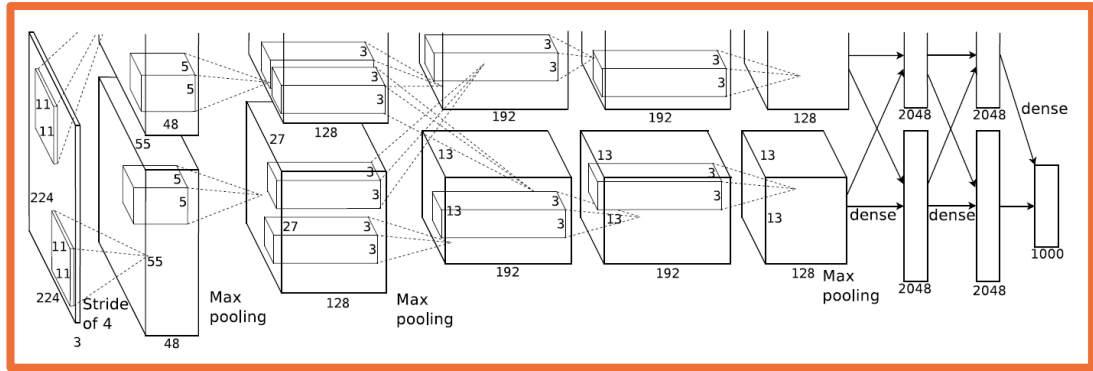
[13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1

[6x6x256] **MAX POOL3**: 3x3 filters at stride 2

[4096] **FC6**: 4096 neurons

[4096] **FC7**: 4096 neurons

[1000] **FC8**: 1000 neurons (class scores)



## Key aspects:

- ReLU instead of sigmoid or tanh
- Specialized normalization layers
- PCA-based data augmentation
- Dropout
- Ensembling

*From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*

# AlexNet – Layers and Key Aspects

INPUT: [224x224x3] memory:  $224 \times 224 \times 3 = 150K$  params: 0 (not counting biases)

CONV3-64: [224x224x64] memory:  $224 \times 224 \times 64 = 3.2M$  params:  $(3 \times 3 \times 3) \times 64 = 1,728$

CONV3-64: [224x224x64] memory:  $224 \times 224 \times 64 = 3.2M$  params:  $(3 \times 3 \times 64) \times 64 = 36,864$

POOL2: [112x112x64] memory:  $112 \times 112 \times 64 = 800K$  params: 0

CONV3-128: [112x112x128] memory:  $112 \times 112 \times 128 = 1.6M$  params:  $(3 \times 3 \times 64) \times 128 = 73,728$

CONV3-128: [112x112x128] memory:  $112 \times 112 \times 128 = 1.6M$  params:  $(3 \times 3 \times 128) \times 128 = 147,456$

POOL2: [56x56x128] memory:  $56 \times 56 \times 128 = 400K$  params: 0

CONV3-256: [56x56x256] memory:  $56 \times 56 \times 256 = 800K$  params:  $(3 \times 3 \times 128) \times 256 = 294,912$

CONV3-256: [56x56x256] memory:  $56 \times 56 \times 256 = 800K$  params:  $(3 \times 3 \times 256) \times 256 = 589,824$

CONV3-256: [56x56x256] memory:  $56 \times 56 \times 256 = 800K$  params:  $(3 \times 3 \times 256) \times 256 = 589,824$

POOL2: [28x28x256] memory:  $28 \times 28 \times 256 = 200K$  params: 0

CONV3-512: [28x28x512] memory:  $28 \times 28 \times 512 = 400K$  params:  $(3 \times 3 \times 256) \times 512 = 1,179,648$

CONV3-512: [28x28x512] memory:  $28 \times 28 \times 512 = 400K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [28x28x512] memory:  $28 \times 28 \times 512 = 400K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params: 0

CONV3-512: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [7x7x512] memory:  $7 \times 7 \times 512 = 25K$  params: 0

FC: [1x1x4096] memory: 4096 params:  $7 \times 7 \times 512 \times 4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params:  $4096 \times 4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params:  $4096 \times 1000 = 4,096,000$

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 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 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

From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition  
 From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

INPUT: [224x224x3] memory:  $224*224*3=150\text{K}$  params: 0 (not counting biases)  
 CONV3-64: [224x224x64] memory:  $224*224*64=3.2\text{M}$  params:  $(3*3*3)*64 = 1,728$   
 CONV3-64: [224x224x64] memory:  $224*224*64=3.2\text{M}$  params:  $(3*3*64)*64 = 36,864$   
 POOL2: [112x112x64] memory:  $112*112*64=800\text{K}$  params: 0  
 CONV3-128: [112x112x128] memory:  $112*112*128=1.6\text{M}$  params:  $(3*3*64)*128 = 73,728$   
 CONV3-128: [112x112x128] memory:  $112*112*128=1.6\text{M}$  params:  $(3*3*128)*128 = 147,456$   
 POOL2: [56x56x128] memory:  $56*56*128=400\text{K}$  params: 0  
 CONV3-256: [56x56x256] memory:  $56*56*256=800\text{K}$  params:  $(3*3*128)*256 = 294,912$   
 CONV3-256: [56x56x256] memory:  $56*56*256=800\text{K}$  params:  $(3*3*256)*256 = 589,824$   
 CONV3-256: [56x56x256] memory:  $56*56*256=800\text{K}$  params:  $(3*3*256)*256 = 589,824$   
 POOL2: [28x28x256] memory:  $28*28*256=200\text{K}$  params: 0  
 CONV3-512: [28x28x512] memory:  $28*28*512=400\text{K}$  params:  $(3*3*256)*512 = 1,179,648$   
 CONV3-512: [28x28x512] memory:  $28*28*512=400\text{K}$  params:  $(3*3*512)*512 = 2,359,296$   
 CONV3-512: [28x28x512] memory:  $28*28*512=400\text{K}$  params:  $(3*3*512)*512 = 2,359,296$   
 POOL2: [14x14x512] memory:  $14*14*512=100\text{K}$  params: 0  
 CONV3-512: [14x14x512] memory:  $14*14*512=100\text{K}$  params:  $(3*3*512)*512 = 2,359,296$   
 CONV3-512: [14x14x512] memory:  $14*14*512=100\text{K}$  params:  $(3*3*512)*512 = 2,359,296$   
 CONV3-512: [14x14x512] memory:  $14*14*512=100\text{K}$  params:  $(3*3*512)*512 = 2,359,296$   
 POOL2: [7x7x512] memory:  $7*7*512=25\text{K}$  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:  $4096*1000 = 4,096,000$

Most memory usage in convolution layers

Most parameters in FC layers

From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition  
 From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

## Parameters and Memory

# Key aspects:

Repeated application of:

- 3x3 conv (stride of 1, padding of 1)
- 2x2 max pooling (stride 2)

Very large number of parameters

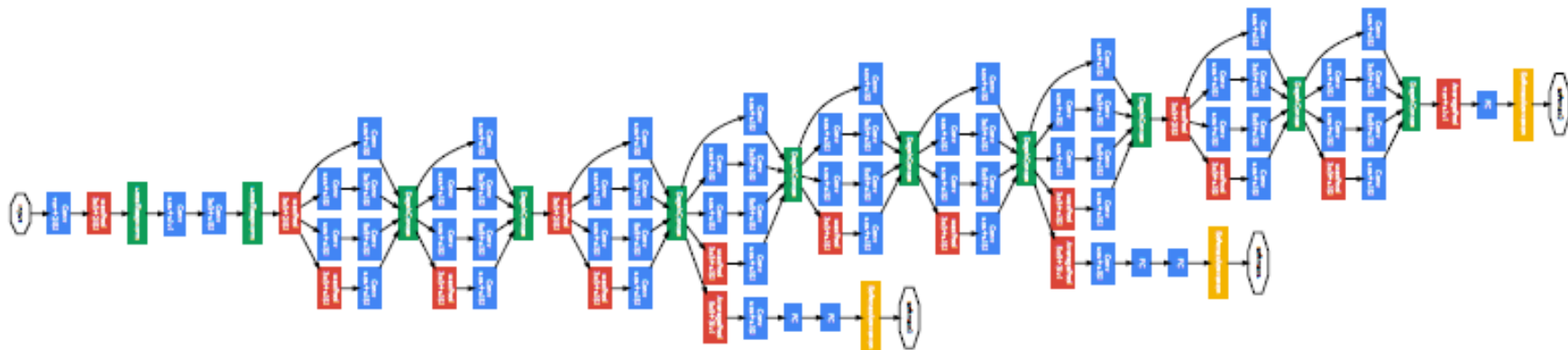
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	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 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 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 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

From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition  
From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

But have become **deeper and more complex**



FC

Conv  
1x1+1(S)

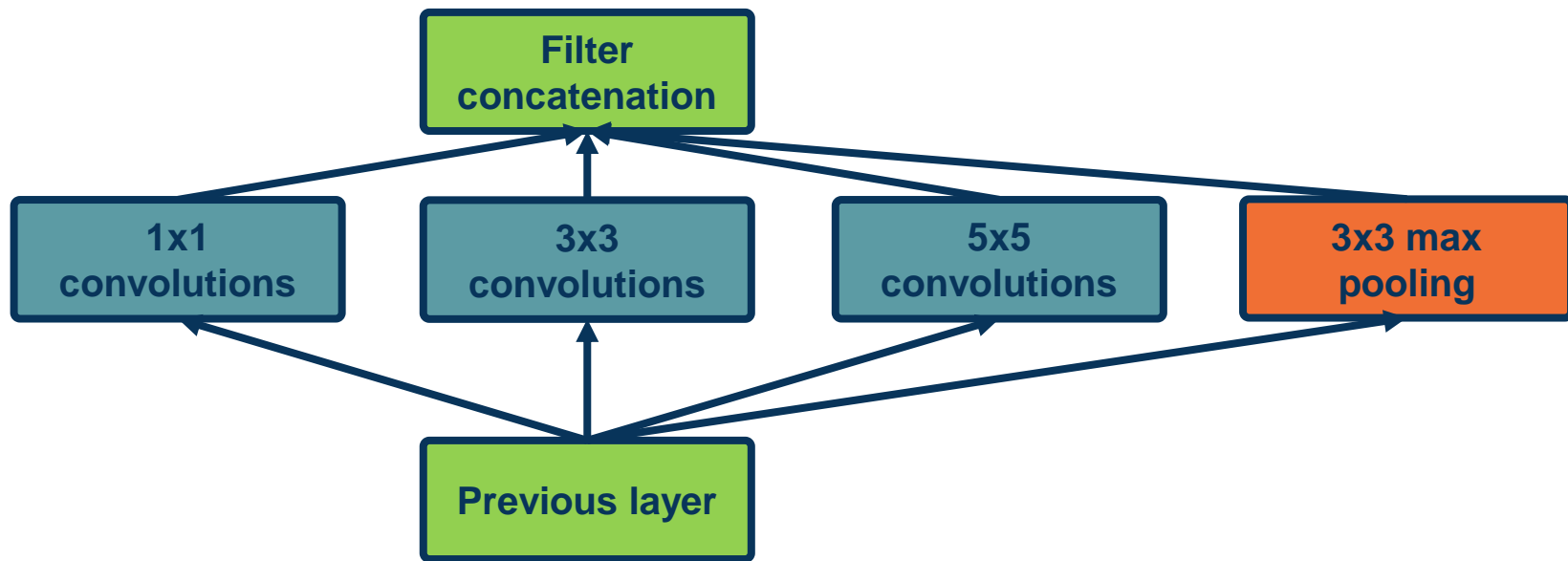
MaxPool  
3x3+1(S)

SoftmaxActivation

From: Szegedy et al. Going deeper with convolutions

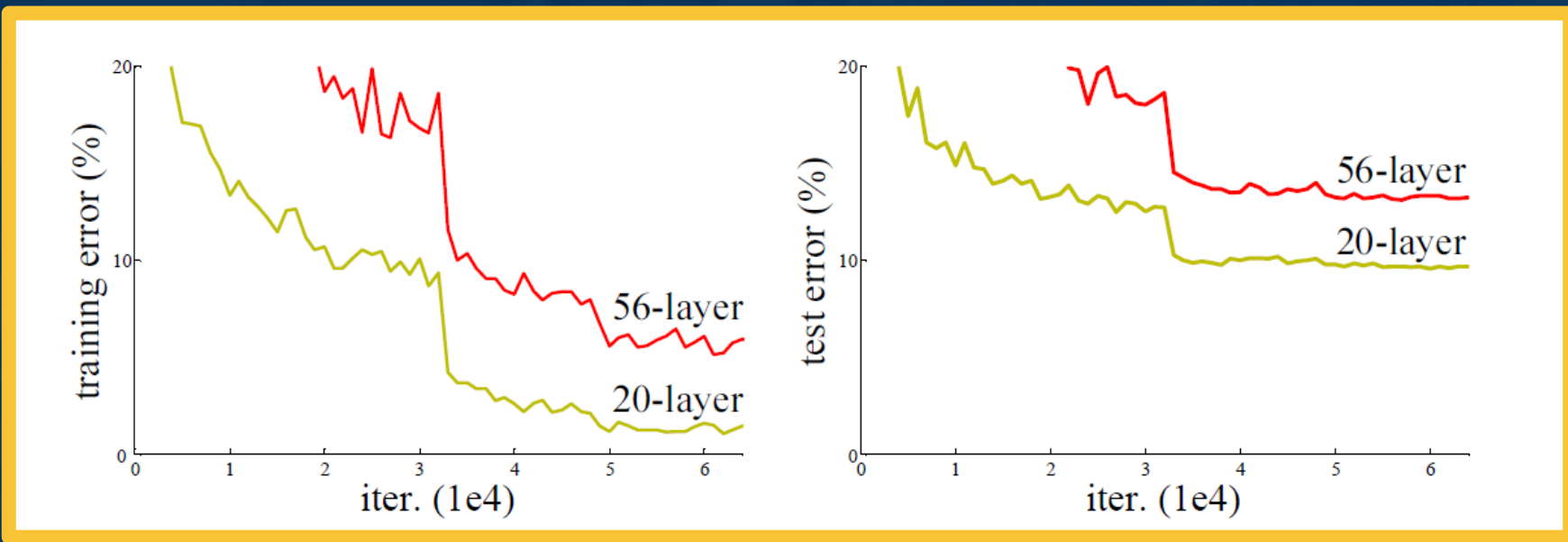
Inception Architecture

**Key idea:** Repeated blocks and multi-scale features



*From: Szegedy et al. Going deeper with convolutions*

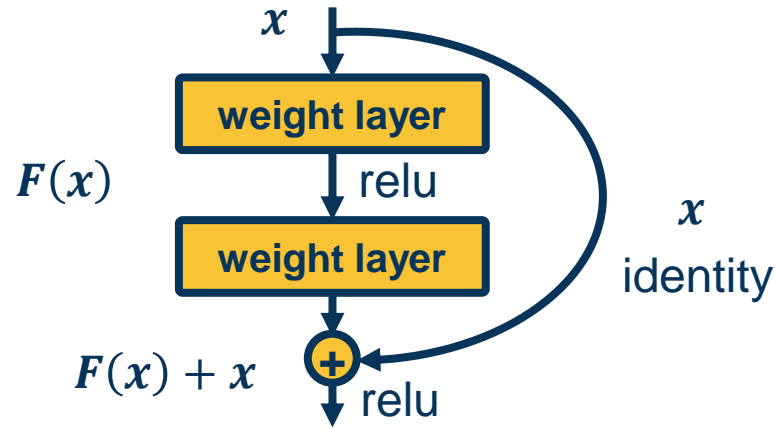
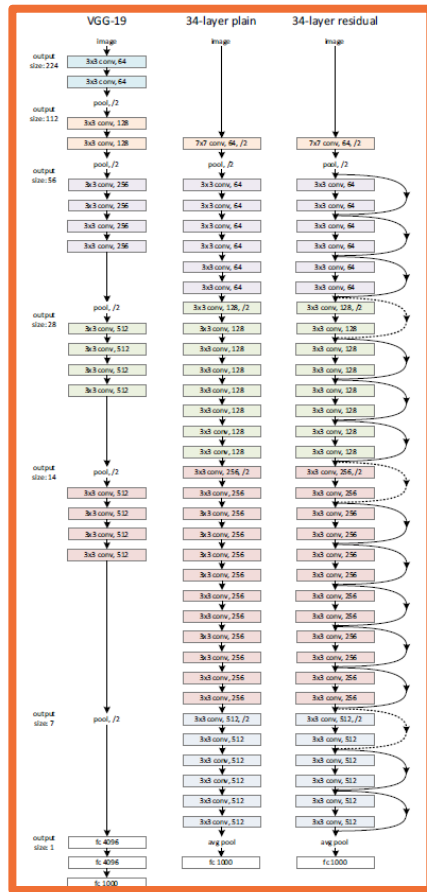
# The Challenge of Depth



*From: He et al., Deep Residual Learning for Image Recognition*

Optimizing very deep networks is challenging!





**Key idea:** Allow information from a layer to propagate to any future layer (forward)

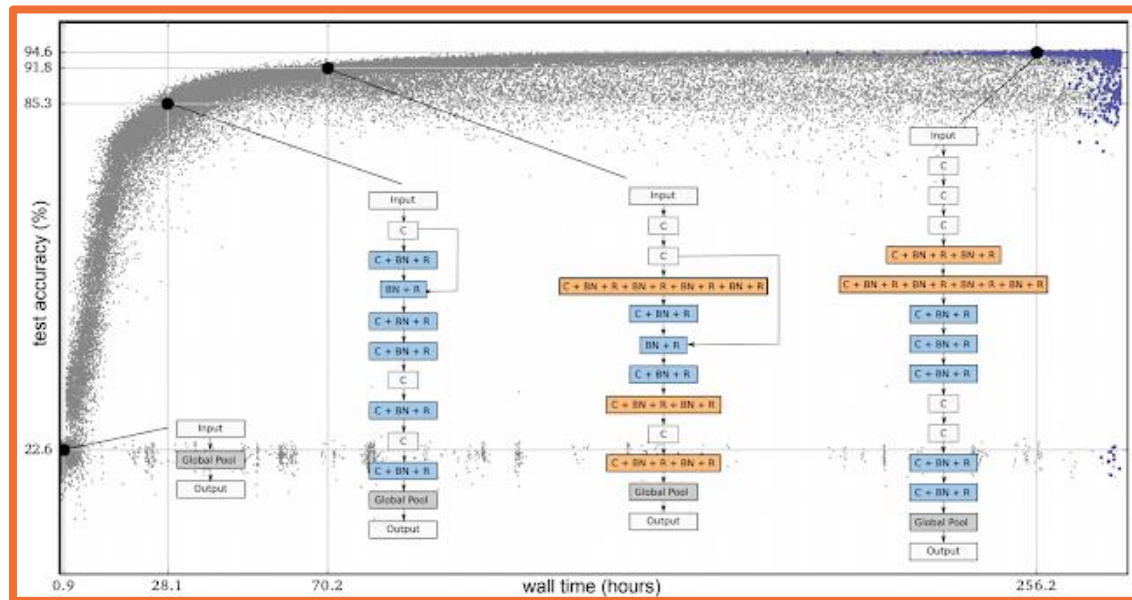
Same is true for gradients!

From: He et al., Deep Residual Learning for Image Recognition



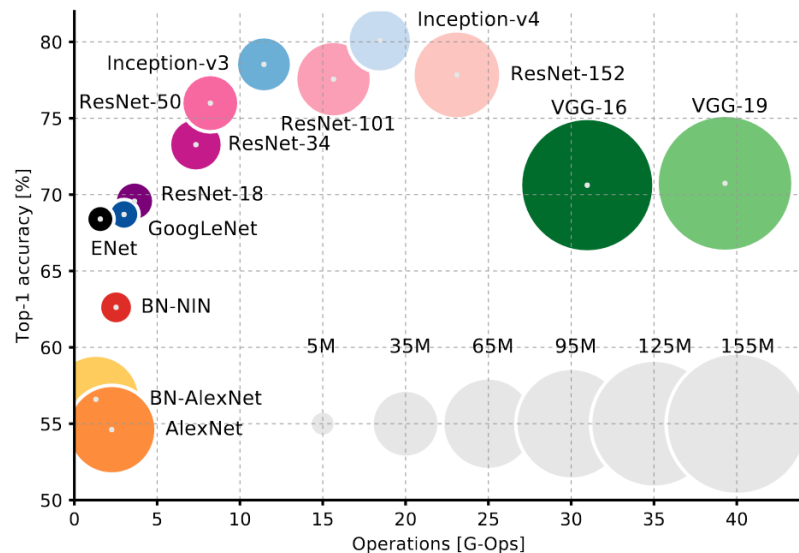
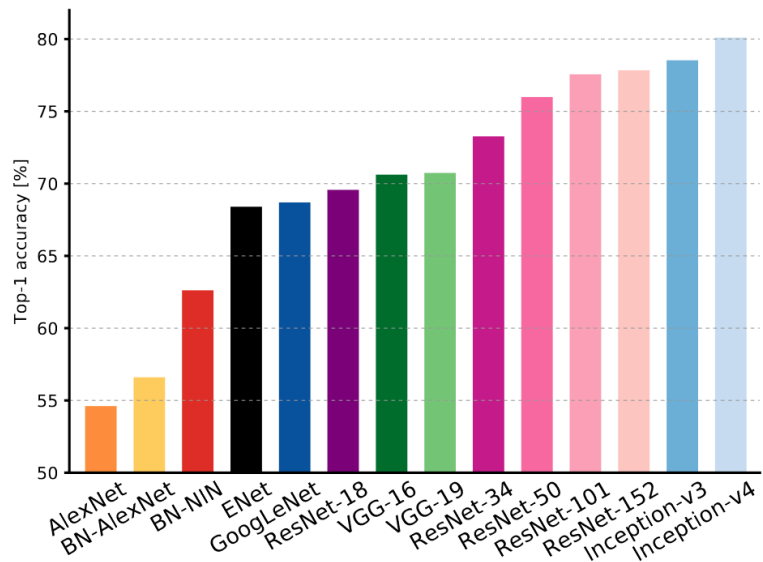
## Several ways to *learn* architectures:

- Evolutionary learning and reinforcement learning
- Prune over-parameterized networks
- Learning of repeated blocks typical



From: <https://ai.googleblog.com/2018/03/using-evolutionary-automl-to-discover.html>

# Computational Complexity



From: An Analysis Of Deep Neural Network Models For Practical Applications