

Topics:

- Convolutional Neural Networks

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

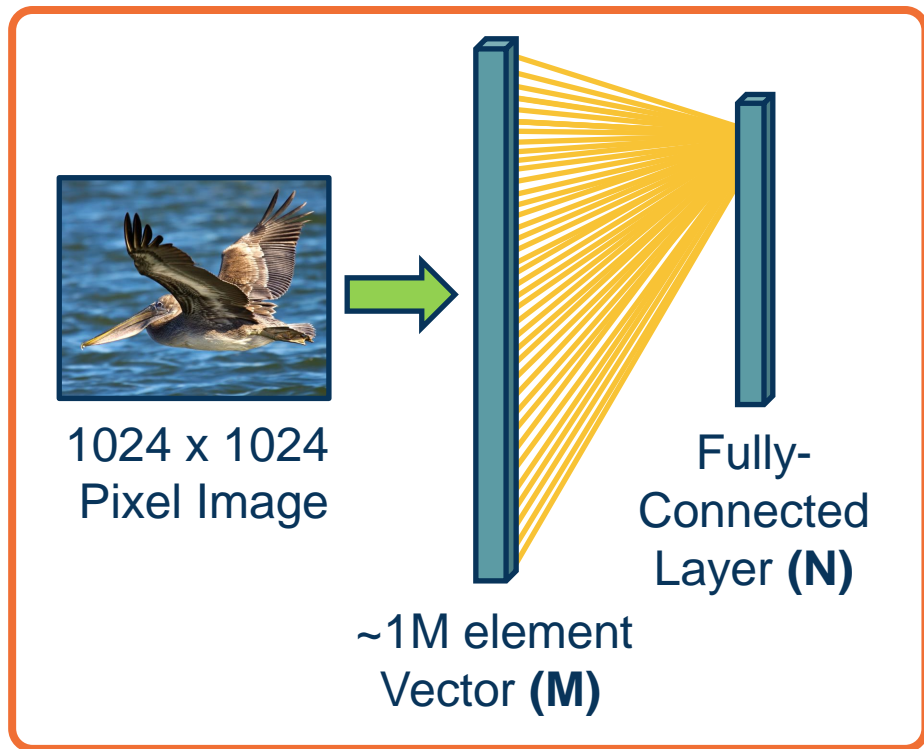
- **Assignment 2**

- 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 (piazza @113)
  - 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))

- **GPU resources**

- **For assignments, can use CPU or Google Colab**
- **Projects:**
  - **Google Cloud Credits**
  - **Working on Amazon AWS**

# The connectivity in linear layers **doesn't** always make sense



How many parameters?

⬡  $M*N$  (weights) +  $N$  (bias)

Hundreds of millions of  
parameters **for just one layer**

**More parameters => More  
data needed**

**Is this necessary?**

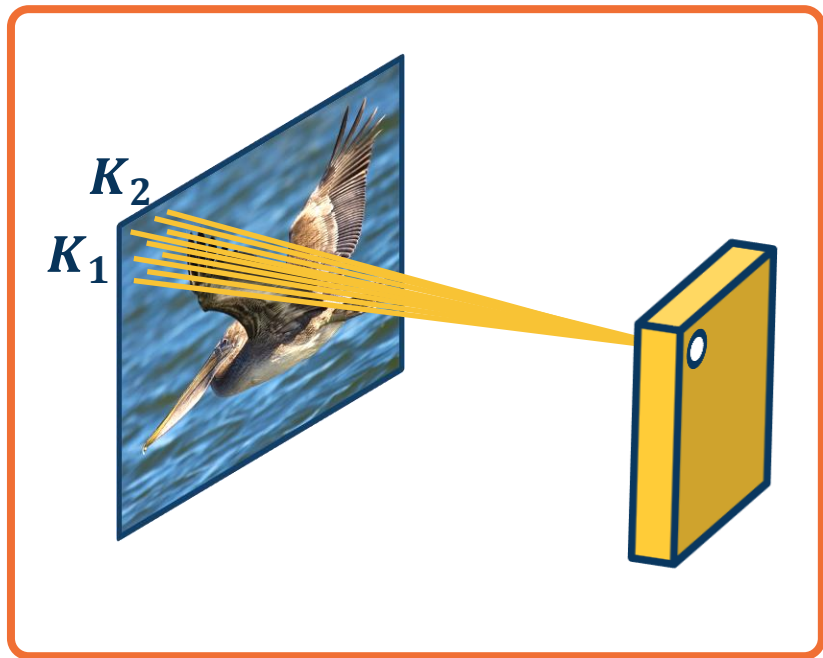
**Limitation of Linear Layers**

## Image features are spatially localized!

- Smaller features repeated across the image
  - Edges
  - Color
  - Motifs (corners, etc.)
- No reason to believe one feature tends to appear in one location vs. another (stationarity)



Can we induce a *bias* in the design of a neural network layer to reflect this?



Each node only receives input from  $K_1 \times K_2$  window (image patch)

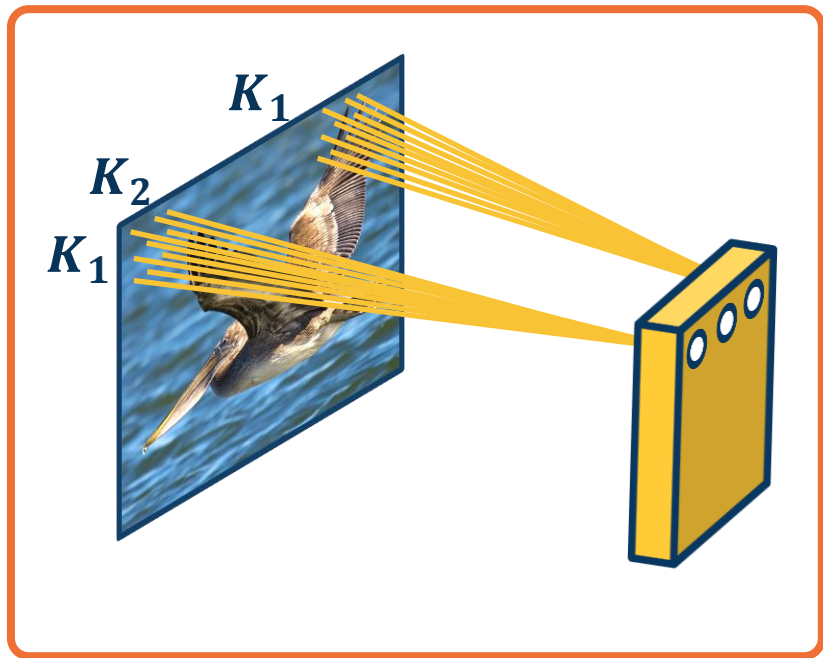
- Region from which a node receives input from is called its **receptive field**

**Advantages:**

- Reduce parameters to  $(K_1 \times K_2 + 1) * N$  where  $N$  is number of output nodes
- Explicitly maintain spatial information

**Do we need to learn location-specific features?**

**Idea 1: Receptive Fields**



Nodes in different locations can **share** features

- No reason to think same feature (e.g. edge pattern) can't appear elsewhere
- Use same weights/parameters in computation graph (**shared weights**)

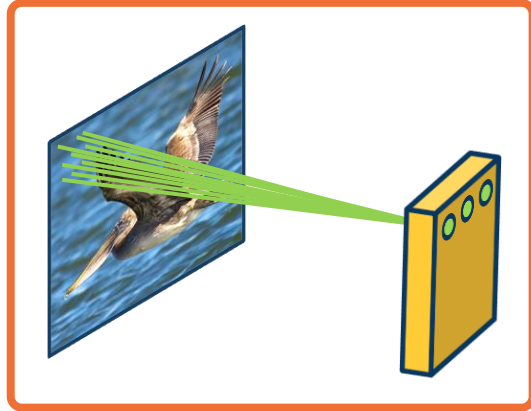
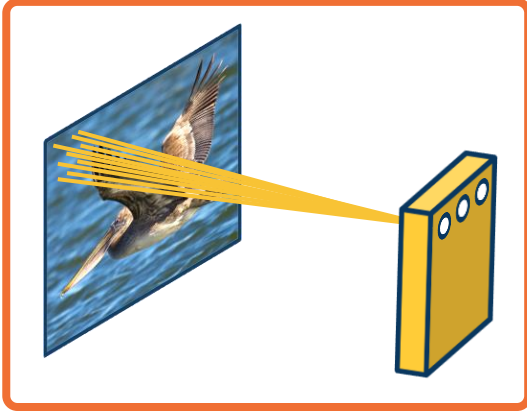
**Advantages:**

- Reduce parameters to  $(K_1 \times K_2 + 1)$
- Explicitly maintain spatial information

## Idea 2: Shared Weights

We can learn **many** such features for this one layer

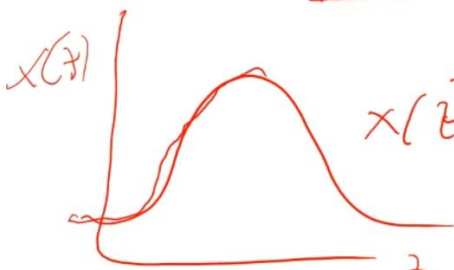
- Weights are **not** shared across different feature extractors
- Parameters:**  $(K_1 \times K_2 + 1) * M$  where  $M$  is number of features we want to learn



## Idea 3: Learn Many Features

This operation is **extremely common** in electrical/computer engineering!

$x(\tau)$     $w(\tau)$     $y(\tau)$

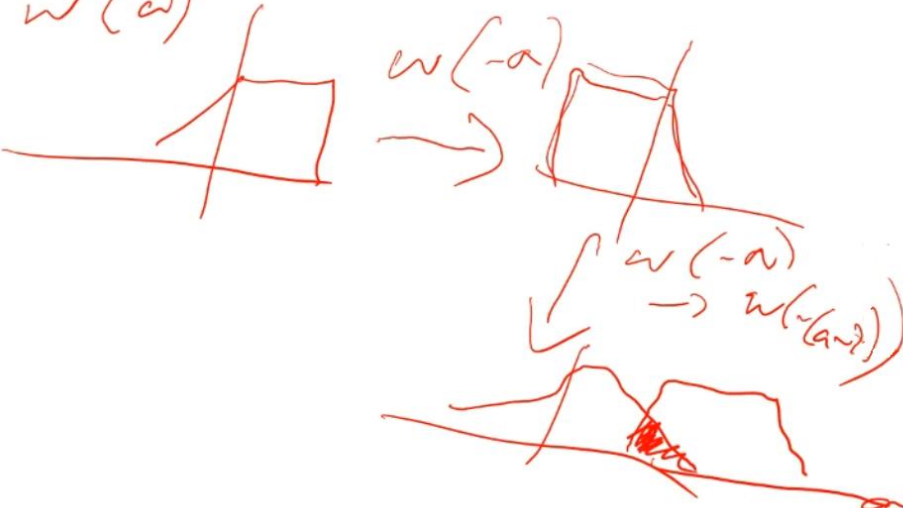


$x(t) = e^{-\frac{(t-t_0)^2}{2\sigma^2}}$

$y(\tau) = (x \ast w)(\tau)$   
 $= \int_{-\infty}^{\tau} x(\tau-a) w(a) da$   
 $= (w \ast x)(\tau) = \int_{-\infty}^{\tau} x(a) w(\tau-a) da$

$y(\tau) = \int_{-\infty}^{\tau} x(a) w(\tau-a) da$

$w(a)$

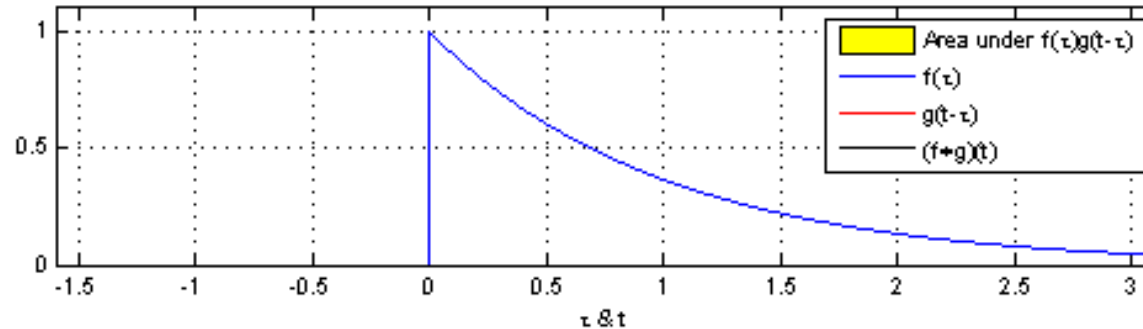
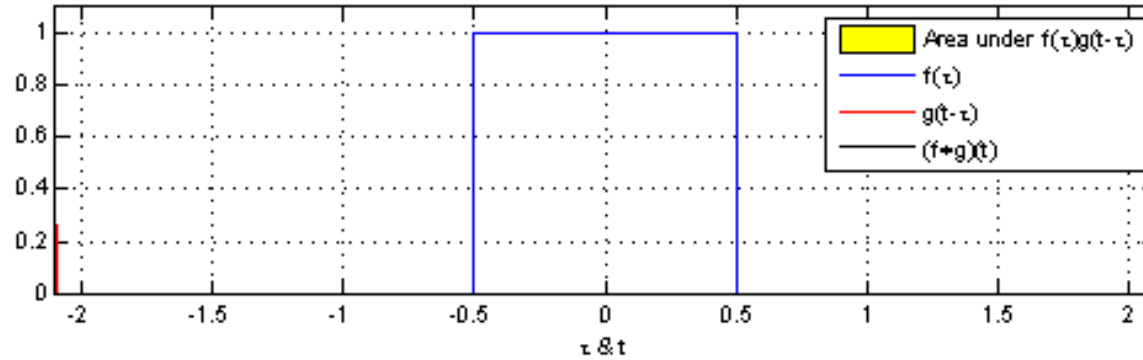


$w(-a)$   
 $\rightarrow w(a-\tau)$

From <https://en.wikipedia.org/wiki/Convolution>



This operation is **extremely common** in electrical/computer engineering!



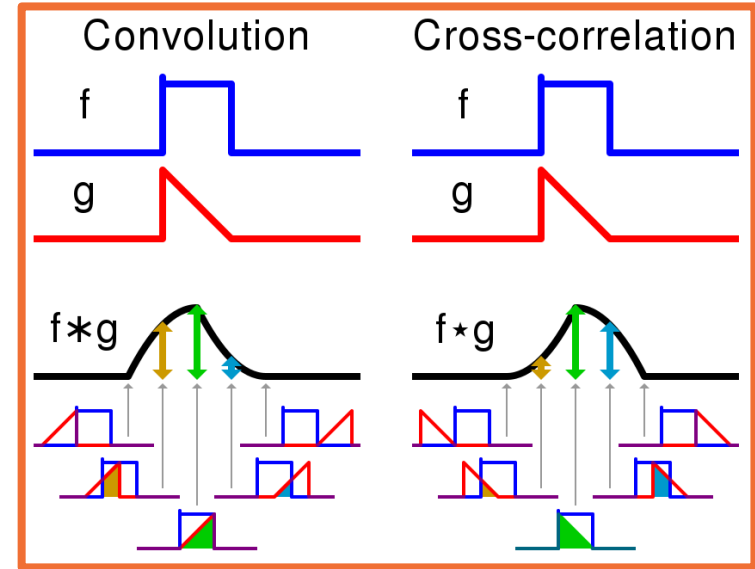
From <https://en.wikipedia.org/wiki/Convolution>

# This operation is **extremely common** in electrical/computer engineering!

In mathematics and, in particular, functional analysis, **convolution** is a mathematical operation on two functions  $f$  and  $g$  producing a third function that is typically viewed as a modified version of one of the original functions, giving the area overlap between the two functions as a function of the amount that one of the original functions is translated.

Convolution is similar to **cross-correlation**.

It has **applications** that include probability, statistics, computer vision, image and signal processing, electrical engineering, and differential equations.



Visual comparison of **convolution** and **cross-correlation**.

From <https://en.wikipedia.org/wiki/Convolution>

**Notation:**  $F \otimes (G \otimes I) = (F \otimes G) \otimes I$

## 1D Convolution

$$y_k = \sum_{n=0}^{N-1} h_n \cdot x_{k-n}$$

$$y_0 = h_0 \cdot x_0$$

$$y_1 = h_1 \cdot x_0 + h_0 \cdot x_1$$

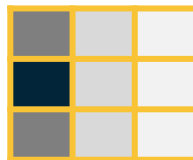
$$y_2 = h_2 \cdot x_0 + h_1 \cdot x_1 + h_0 \cdot x_2$$

$$y_3 = h_3 \cdot x_0 + h_2 \cdot x_1 + h_1 \cdot x_2 + h_0 \cdot x_3$$

$$\vdots$$

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

## 2D Convolution



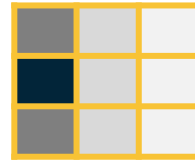
## 2D Convolution

Image



Kernel  
(or filter)

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



Output /  
filter /  
feature map



We will make this convolution operation **a layer** in the neural network

- Initialize kernel values randomly and optimize them!
- These are our parameters (plus a bias term per filter)

## 2D Convolution

Image



Kernel  
(or filter)

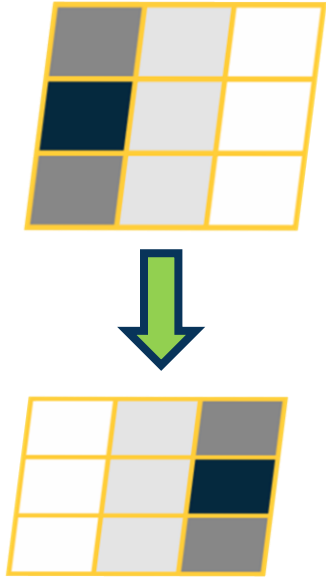
$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



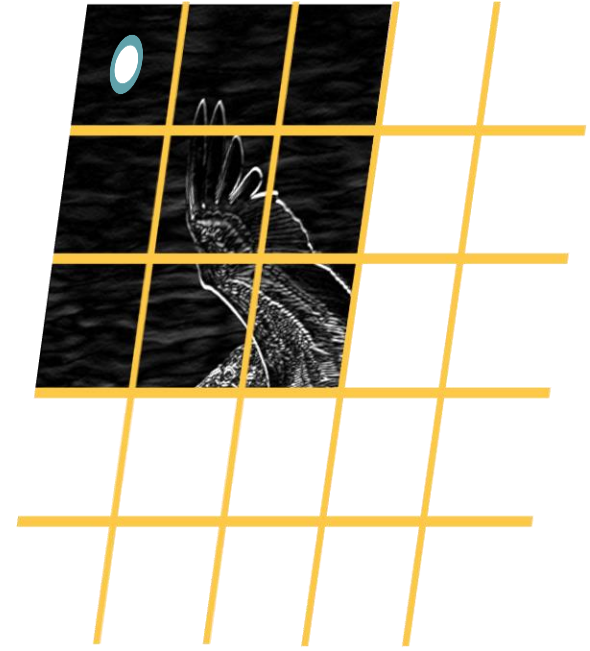
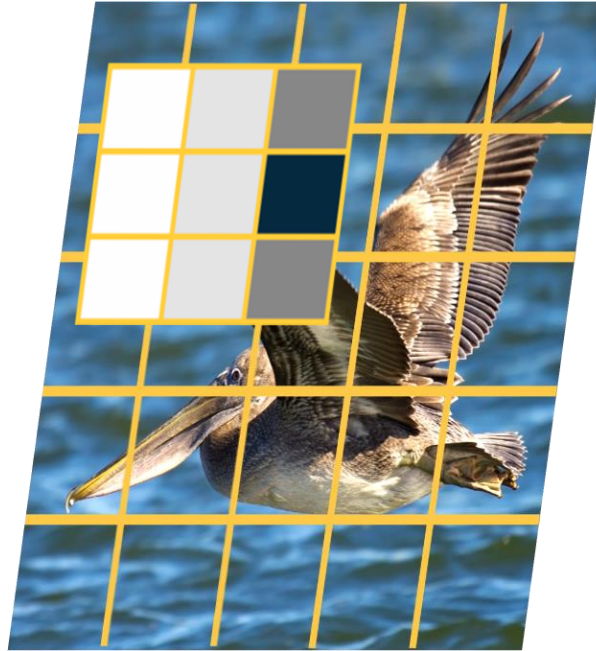
Output /  
filter /  
feature map



1. Flip kernel  
(rotate 180  
degrees)



2. Stride  
along image



The Intuitive Explanation

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

$$\left( -\frac{H-1}{2}, -\frac{W-1}{2} \right)$$



$W = 5$

$$\left( \frac{H-1}{2}, \frac{W-1}{2} \right)$$

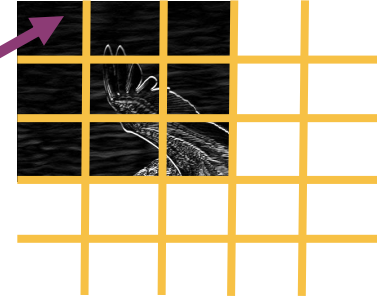
$(0, 0)$

$k_1 = 3$



$k_2 = 3$

$(k_1 - 1, k_2 - 1)$

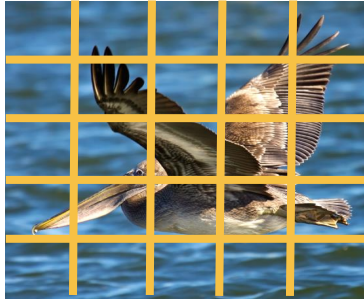


$$y(0, 0) = x(-2, -2)k(2, 2) + x(-2, -1)k(2, 1) + x(-2, 0)k(2, 0) + x(-2, 1)k(2, -1) + x(-2, 2)k(2, -2) + \dots$$

$$y(r, c) = (x * k)(r, c) = \sum_{a=-\frac{K_1-1}{2}}^{\frac{k_1-1}{2}} \sum_{b=-\frac{k_2-1}{2}}^{\frac{k_2-1}{2}} x(r-a, c-b) k(a, b)$$

(0, 0)

$H = 5$



$W = 5$

$(H-1, W-1)$

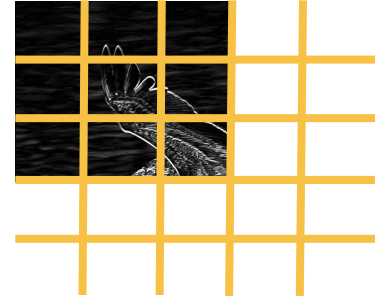
$(-\frac{k_1-1}{2}, -\frac{k_2-1}{2})$

$k_1 = 3$



$k_2 = 3$

$(\frac{k_1-1}{2}, \frac{k_2-1}{2})$



Centering Around the Kernel



As we have seen:

- **Convolution:** Start at end of kernel and move back
- **Cross-correlation:** Start in the beginning of kernel and move forward (same as for image)

An **intuitive interpretation** of the relationship:

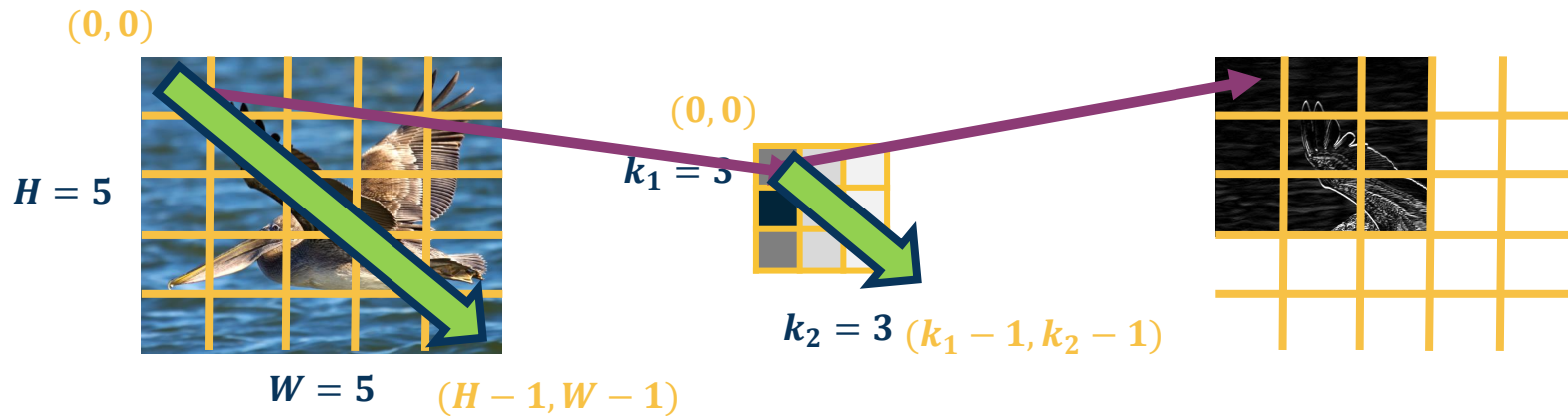
- Take the kernel, and rotate 180 degrees along center (sometimes referred to as “flip”)
- Perform cross-correlation
- (Just dot-product filter with image!)

$$K = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$



$$K' = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

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



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

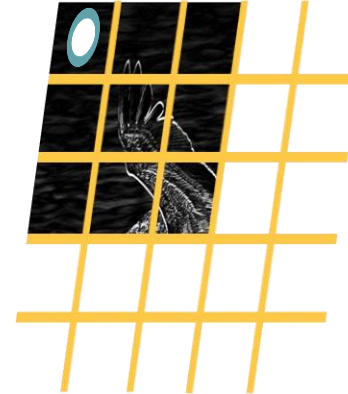
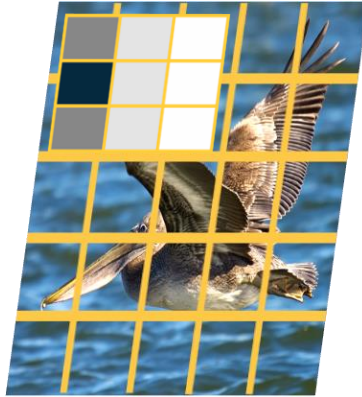
$$x(0:2,0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix}$$

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

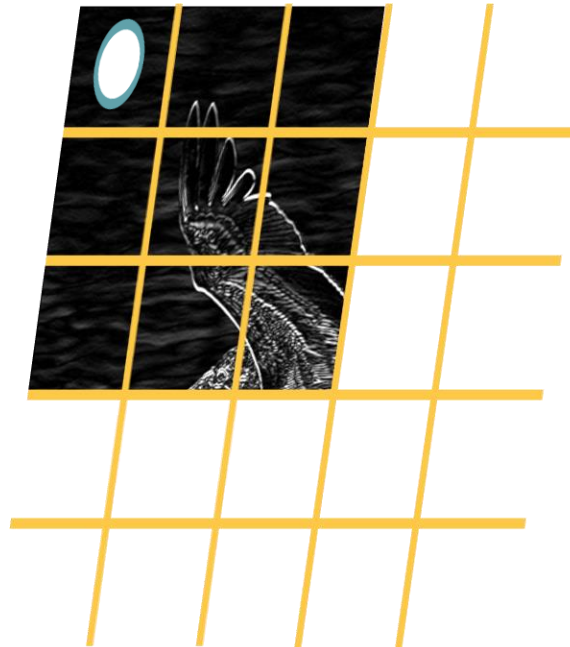
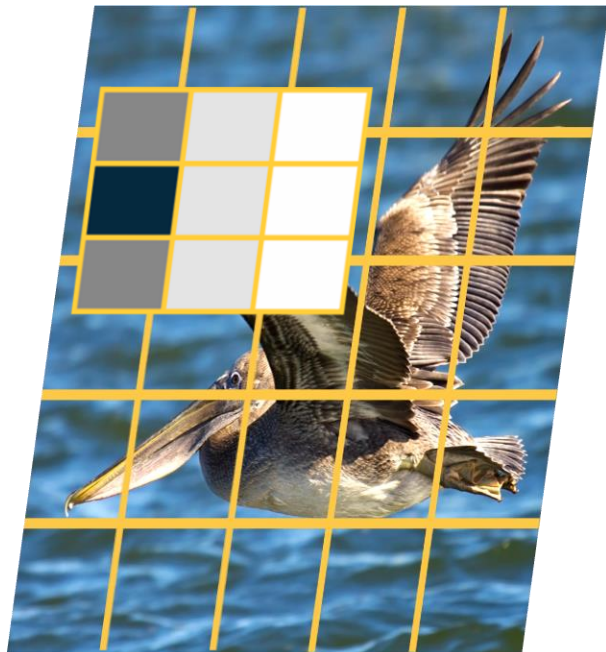


$$x(0:2,0:2) \cdot K' = 65 + \text{bias}$$

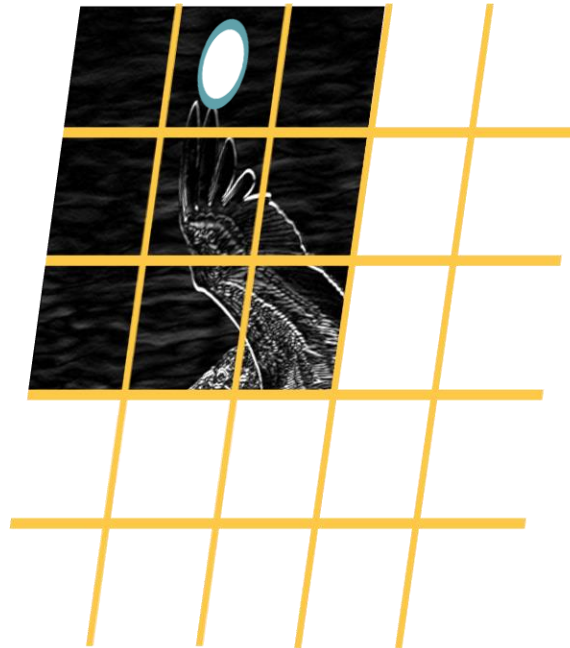
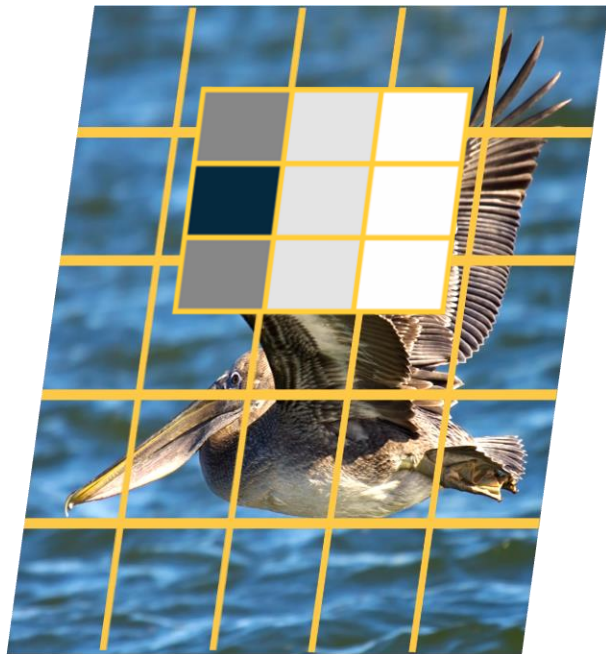
Dot product  
(element-wise multiply and sum)



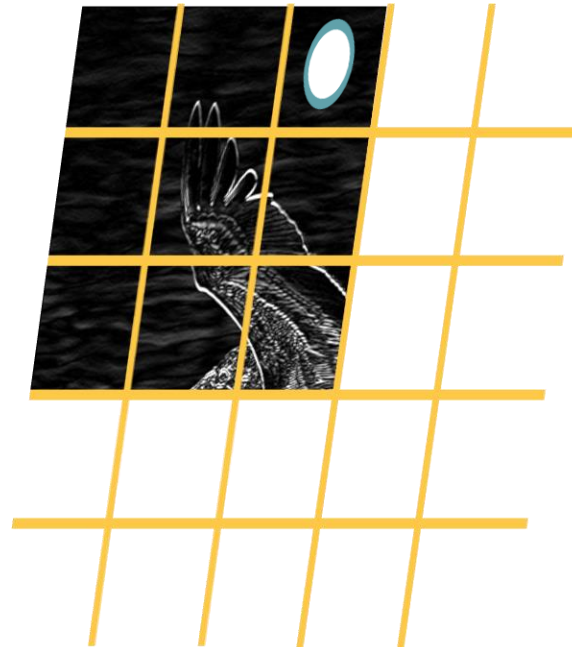
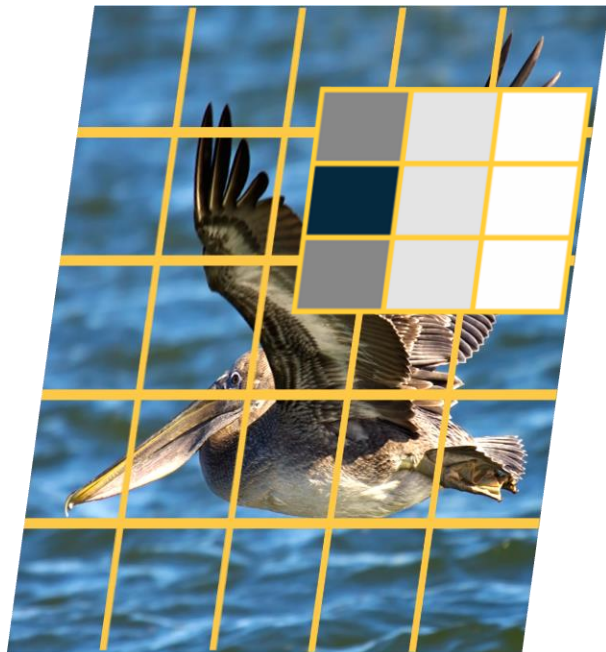
## Cross-Correlation



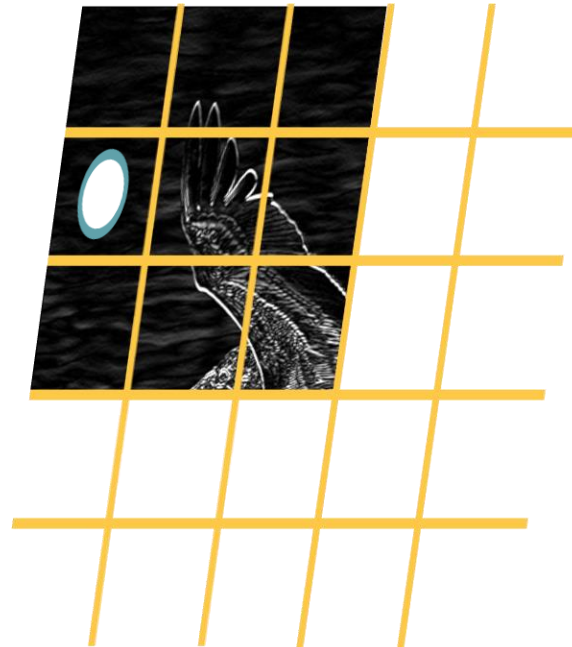
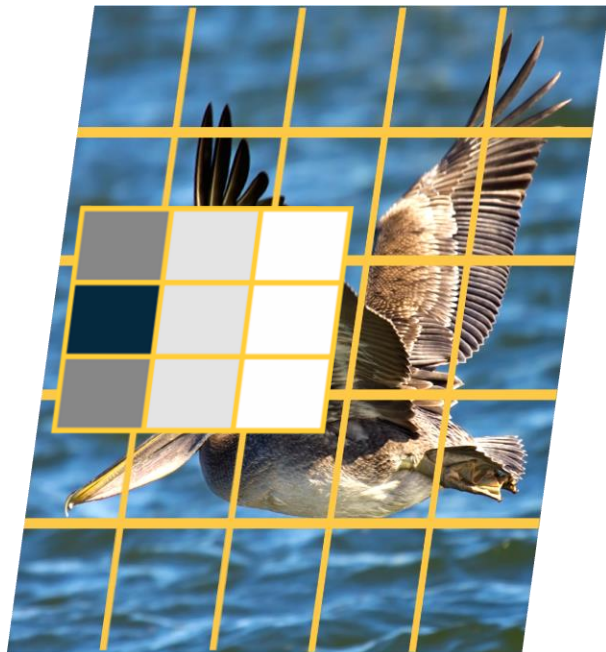
## Convolution and Cross-Correlation



## Convolution and Cross-Correlation

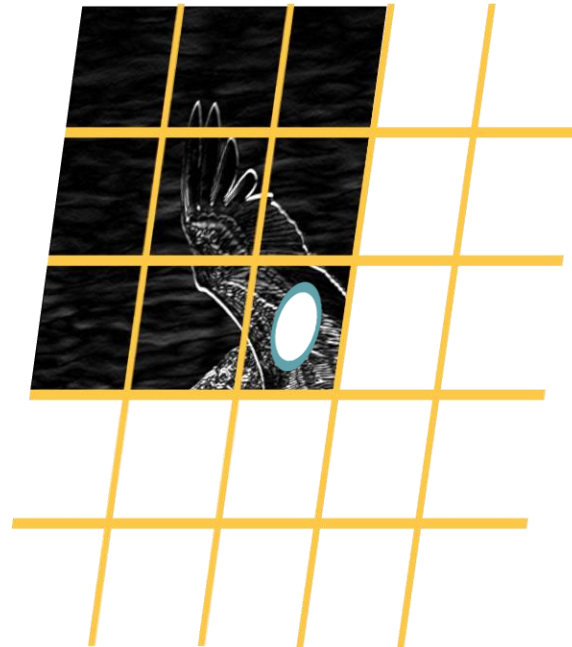
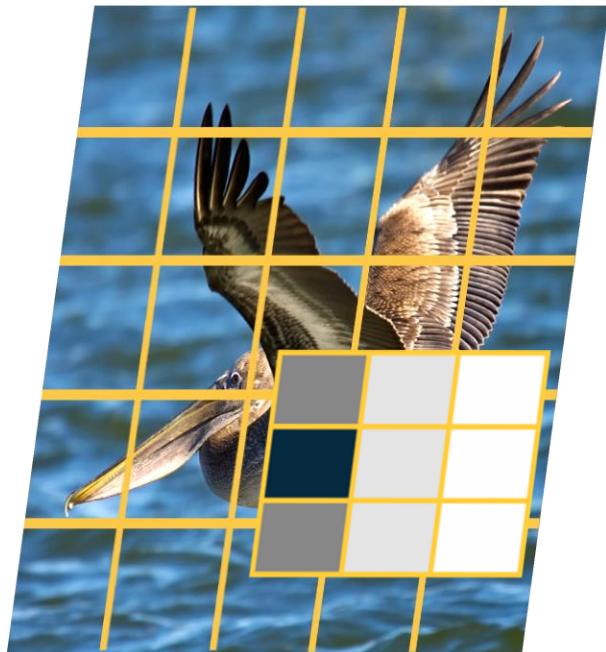


## Convolution and Cross-Correlation



## 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



# **Input & Output Sizes**

# Convolution Layer Hyper-Parameters

## Parameters

- **in\_channels** (*int*) – Number of channels in the input image
- **out\_channels** (*int*) – Number of channels produced by the convolution
- **kernel\_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding\_mode** (*string, optional*) – 'zeros', 'reflect', 'replicate' or 'circular'. Default: 'zeros'

Convolution operations have several hyper-parameters

From: <https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d>

**Output size** of vanilla convolution operation is  $(H - k_1 + 1) \times (W - k_2 + 1)$

🟡 This is called a “**valid**” **convolution** and only applies kernel within image

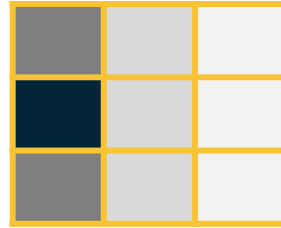
$(0, 0)$



$W = 5$   $(H - 1, W - 1)$

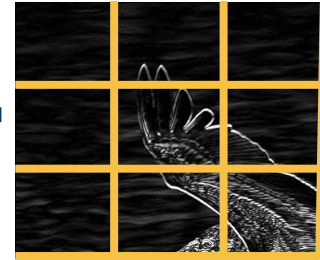
$(0, 0)$

$k_1 = 3$



$k_2 = 3$   $(k_1 - 1,$   
 $k_2 - 1)$

$H - k_1 + 1$

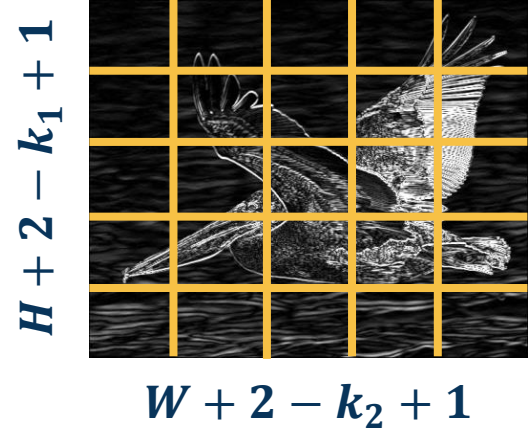
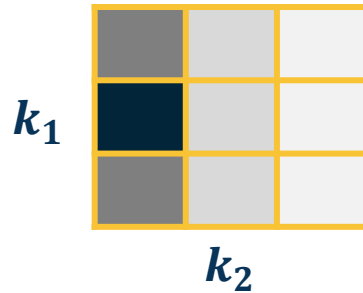
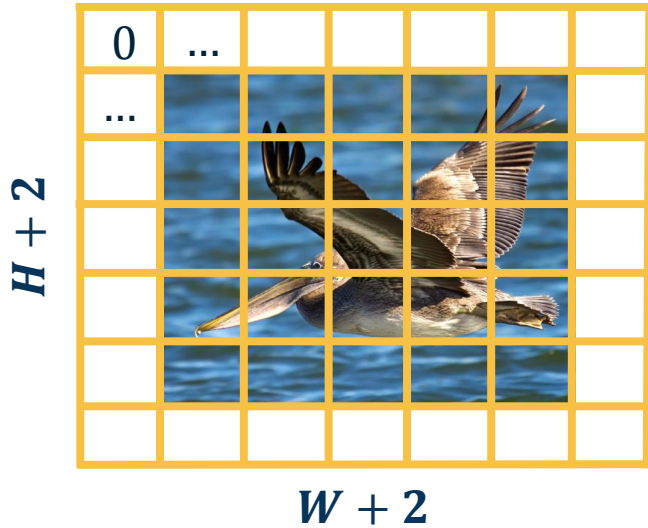


$W - k_2 + 1$

**Valid Convolution**

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

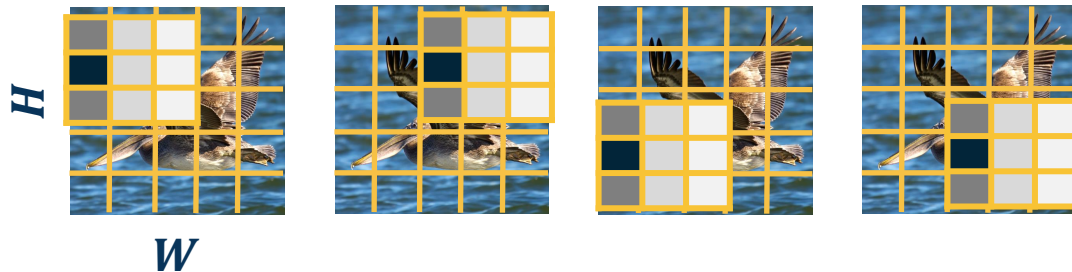
- Zeros, mirrored image, etc.
- Note padding often refers to pixels added to **one size** ( $P = 1$  here)



We can move the filter along the image using larger steps (**stride**)

- This can potentially result in **loss of information**
- Can be used for **dimensionality reduction** (not recommended)

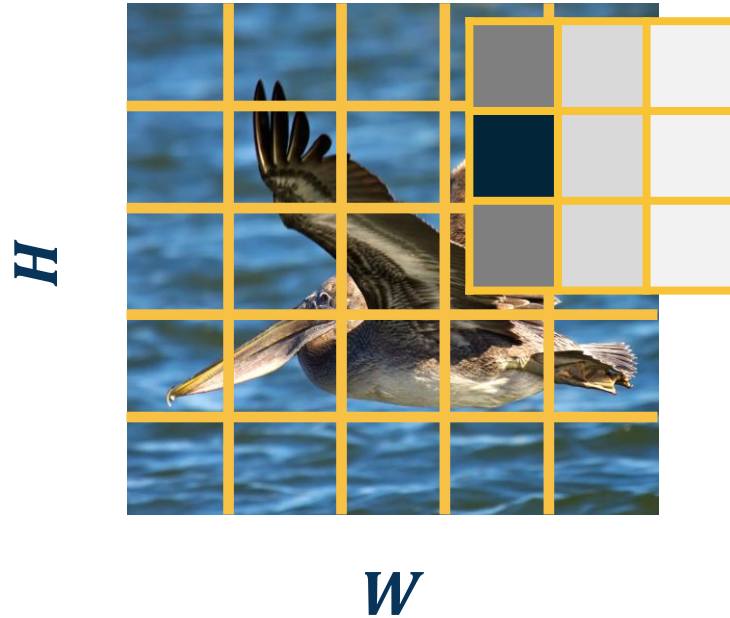
**Stride = 2 (every other pixel)**



$$\begin{matrix} (H - k_1)/2 + 1 \\ (W - k_2)/2 + 1 \end{matrix}$$

The diagram shows a 4x4 grid of bird images with a 3x3 yellow filter. The filter is positioned at (1,1) in a 1-indexed grid. The vertical axis is labeled  $(H - k_1)/2 + 1$  and the horizontal axis is labeled  $(W - k_2)/2 + 1$ .

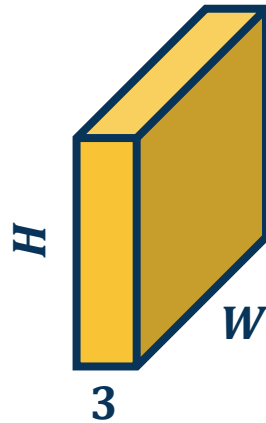
Stride can result in **skipped pixels**, e.g. stride of 3 for 5x5 input



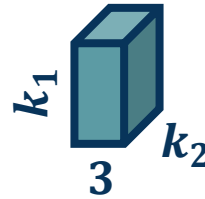
Invalid Stride

We have shown inputs as a **one-channel image** but in reality they have **three channels** (red, green, blue)

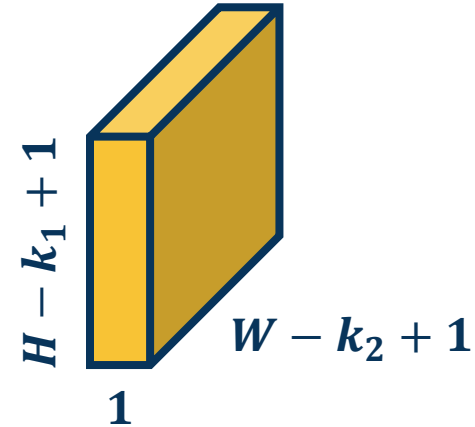
🟡 In such cases, we have **3-channel kernels**!



Image



Kernel

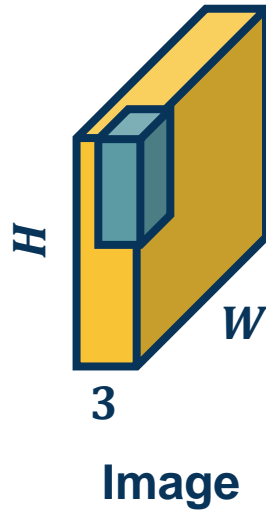


Feature Map



We have shown inputs as a **one-channel image** but in reality they have three channels (red, green, blue)

✧ In such cases, we have **3-channel kernels**!



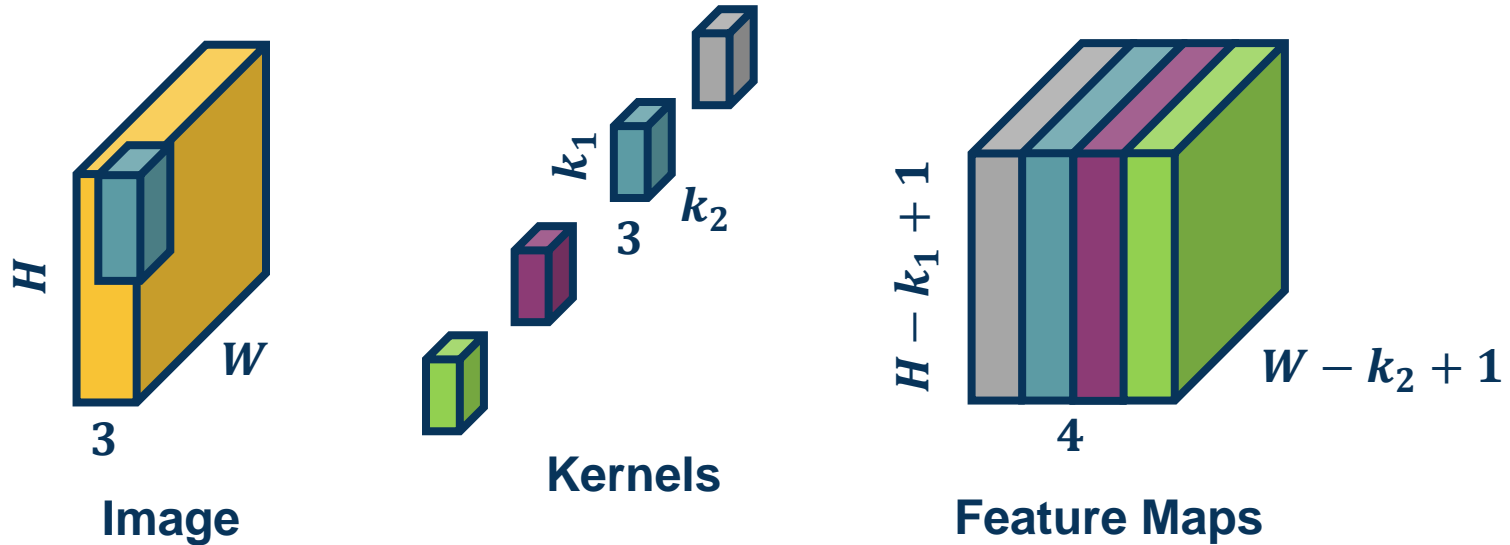
Similar to before, we perform **element-wise multiplication** between kernel and image patch, summing them up (**dot product**)

✧ Except with  $k_1 * k_2 * 3$  values

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*

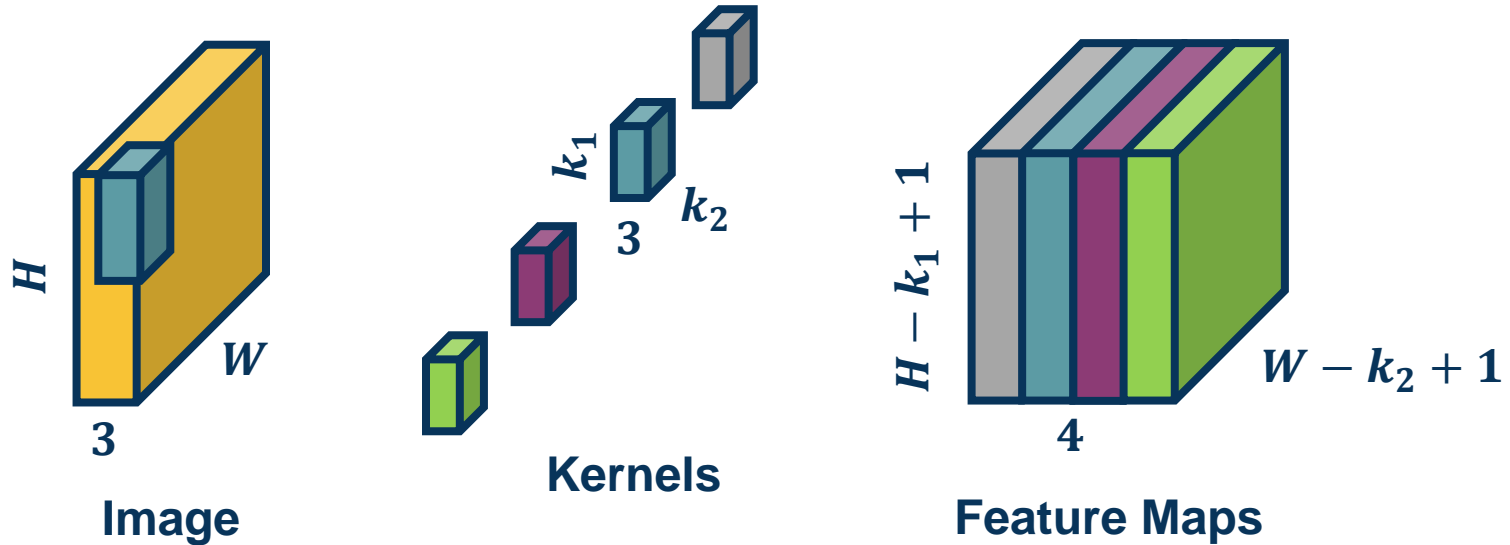


## Multiple Kernels

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

Example:

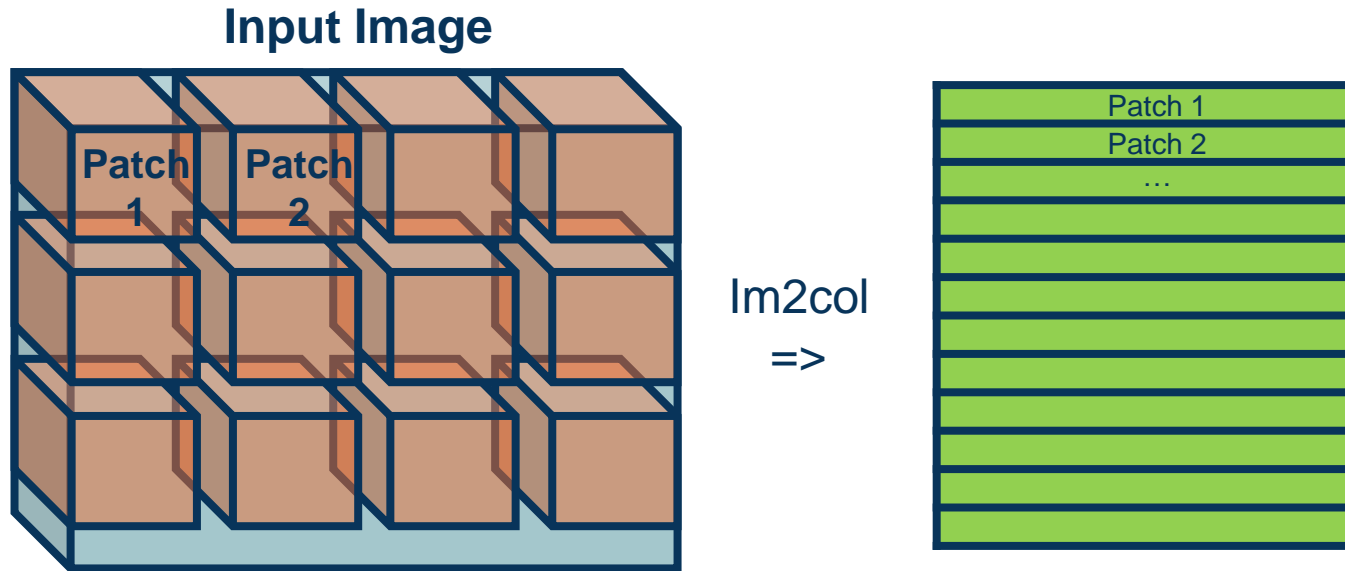
$k_1 = 3, k_2 = 3, N = 4$  *input channels* = 3, then  $(3 * 3 * 3 + 1) * 4 = 112$



Number of Parameters

Just as before, in practice we can **vectorize** this operation

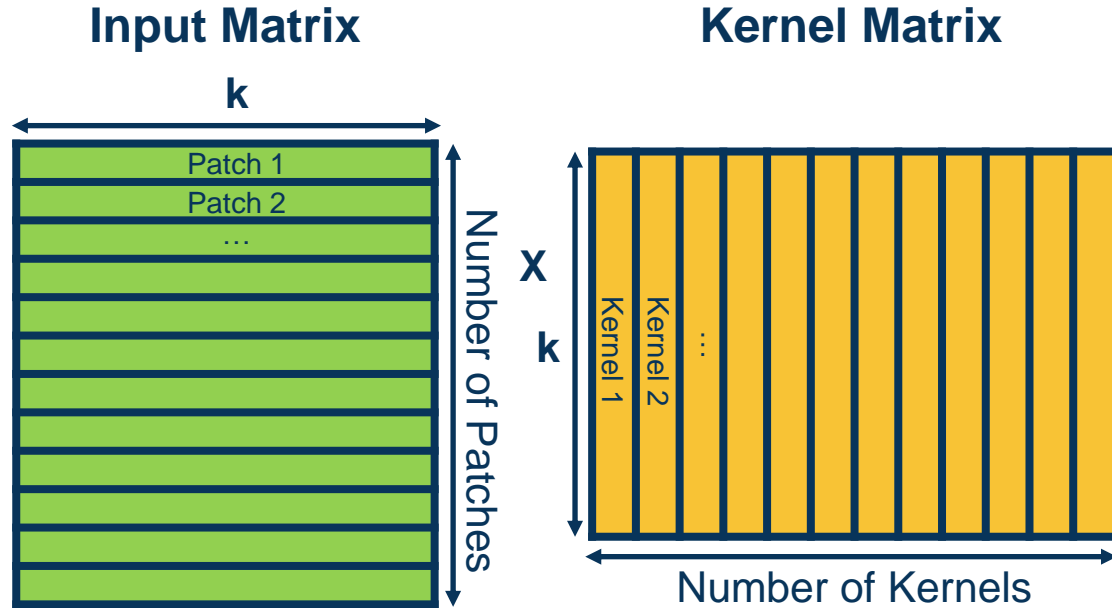
- **Step 1:** Lay out image patches in vector form (note can overlap!)



Adapted from: <https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/>

Just as before, in practice we can **vectorize** this operation

🟡 **Step 2:** Multiple patches by kernels



Adapted from: <https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/>

# **Backwards Pass for Convolution Layer**

It is instructive to calculate **the backwards pass** of a convolution layer

- Similar to fully connected layer, will be **simple vectorized linear algebra operation!**
- We will see a **duality** between cross-correlation and convolution

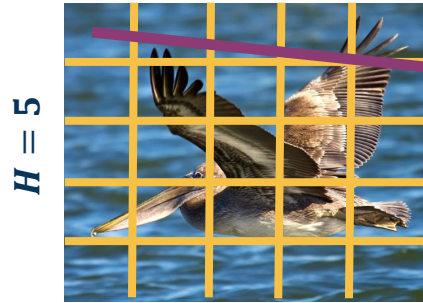
$$K = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$



$$K' = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

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

$(0, 0)$



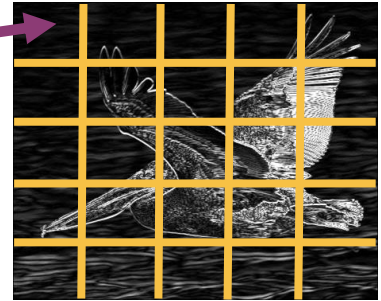
$W = 5$   $(H - 1, W - 1)$

$(0, 0)$



$k_2 = 3$

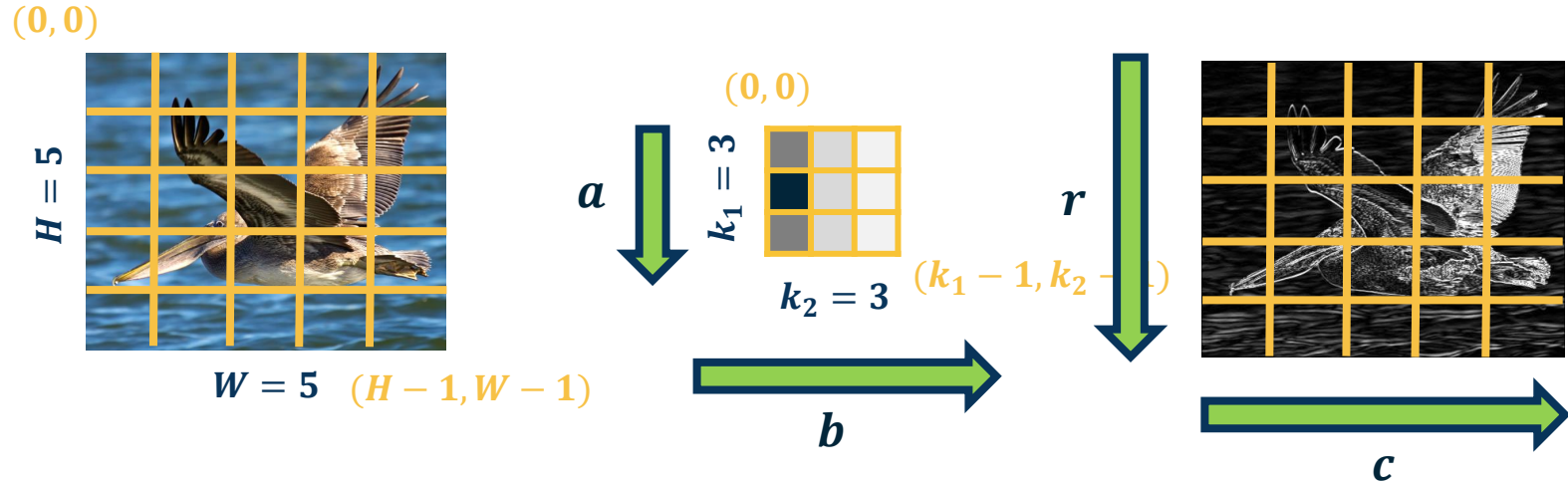
$(k_1 - 1, k_2 - 1)$



Recap: Cross-Correlation



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



**Some simplification:** 1 channel input, 1 kernel (channel output), padding (here 2 pixels on right/bottom) to make output the same size

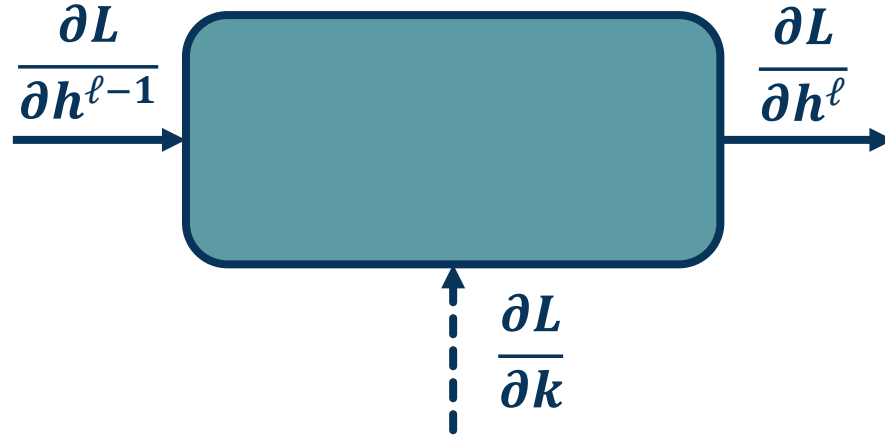
$$y(r, c) = (x * k)(r, c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r + a, c + b) k(a, b)$$

$$|y| = H \times W$$

$$\frac{\partial L}{\partial y} ?$$

Assume size  $H \times W$  (add padding, change convention a bit for convenience)

$$\frac{\partial L}{\partial y(r, c)} \text{ to access element}$$



$$\frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial h^{\ell-1}}$$

**Gradient for passing back**

$$\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial k}$$

**Gradient for weight update**

(weights = k, i.e. kernel values)

**Backpropagation Chain Rule**

# Gradient for Convolution Layer

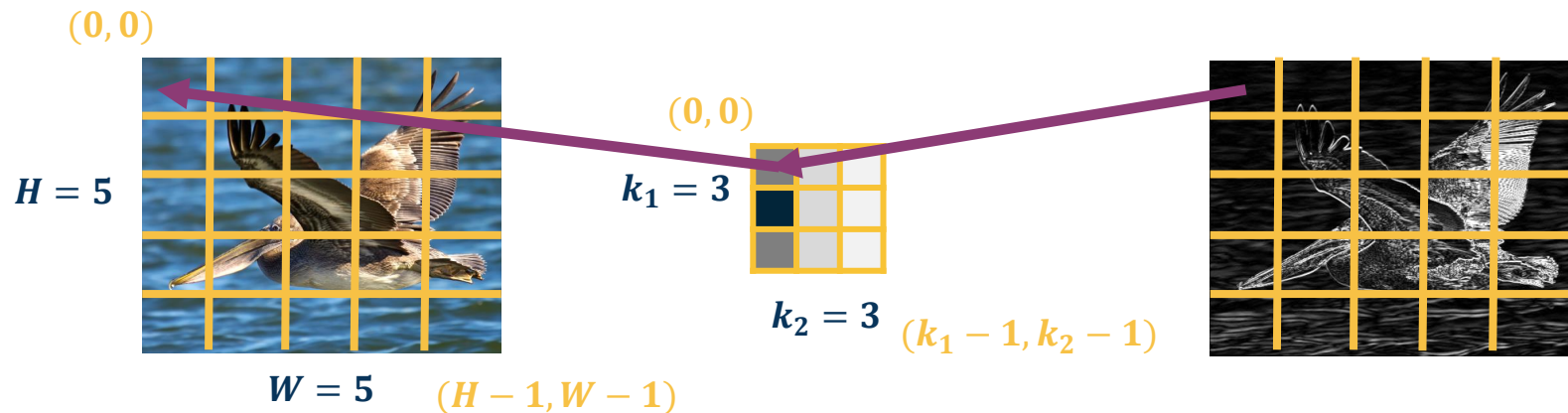
$$\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h^\ell} \frac{\partial h^\ell}{\partial k}$$

What does this weight affect at the output?

Gradient for weight update

Calculate one pixel at a time  $\frac{\partial L}{\partial k(a, b)}$

Everything!



What a Kernel Pixel Affects at Output

Need to incorporate all upstream gradients:

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

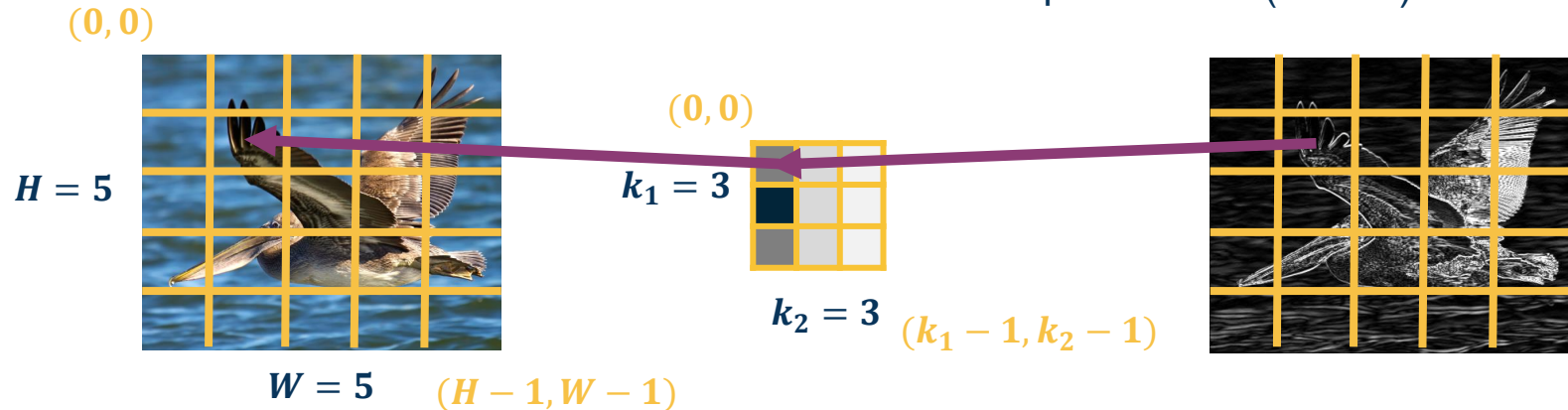
Chain Rule:

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

Sum over  
all output  
pixels

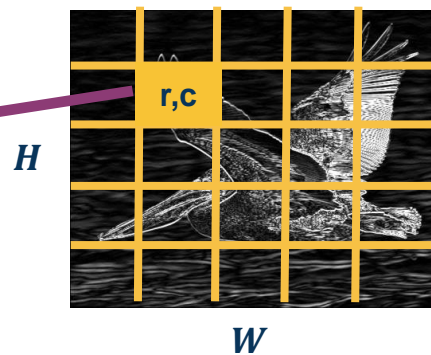
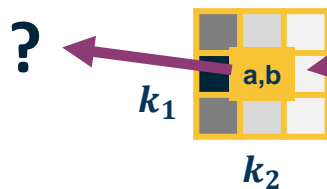
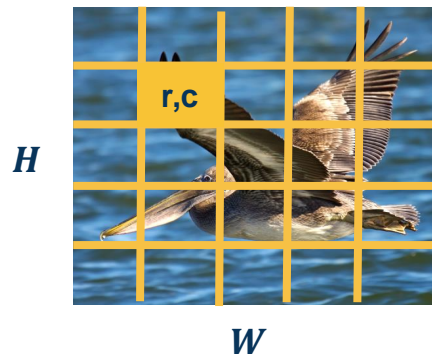
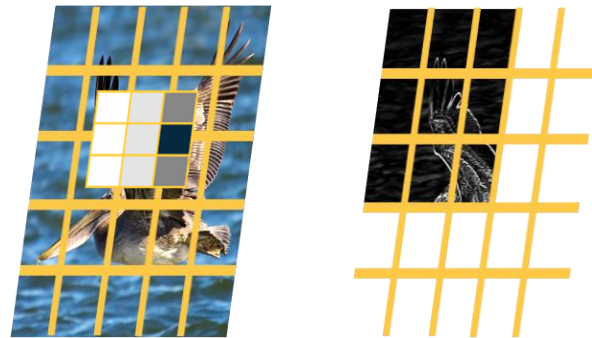
Upstream  
gradient  
(known)

We will  
compute



Chain Rule over all Output Pixels

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



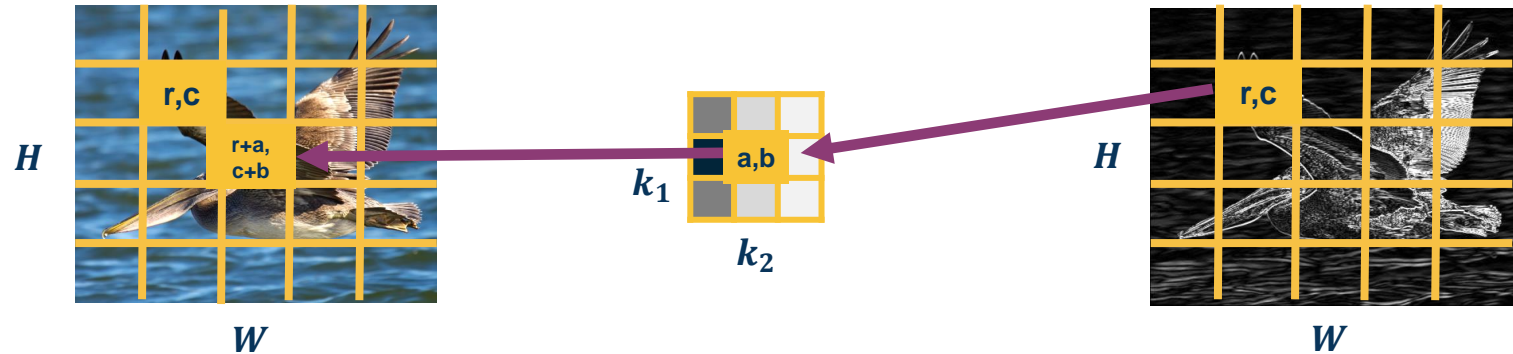
Chain Rule over all Output Pixels

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

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

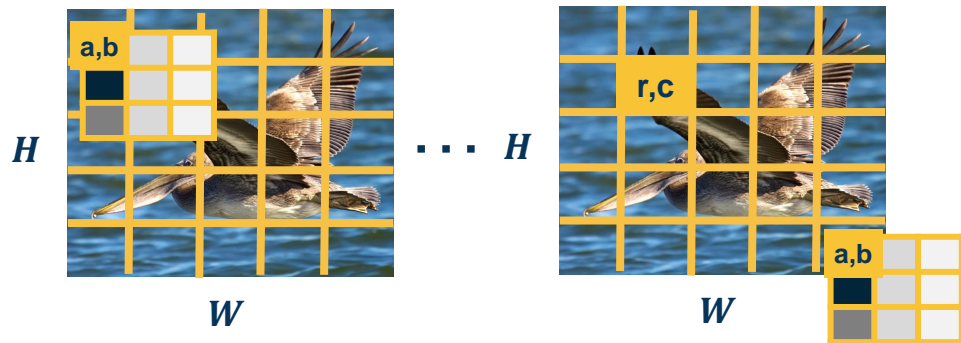
Does this look familiar?

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

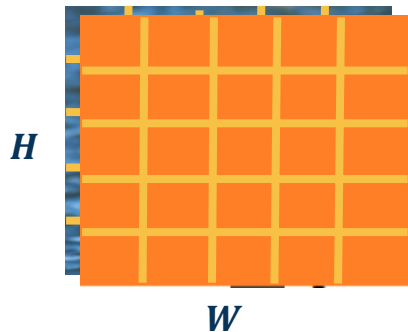




## Forward Pass



## 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)



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

Forward and Backward Duality

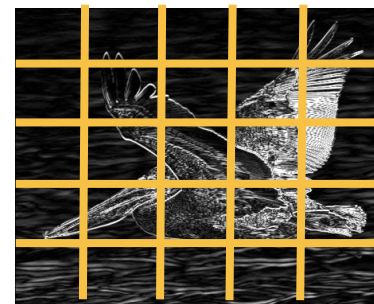
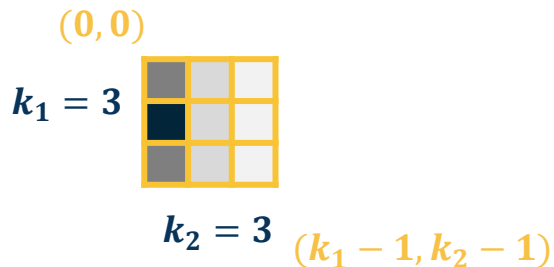
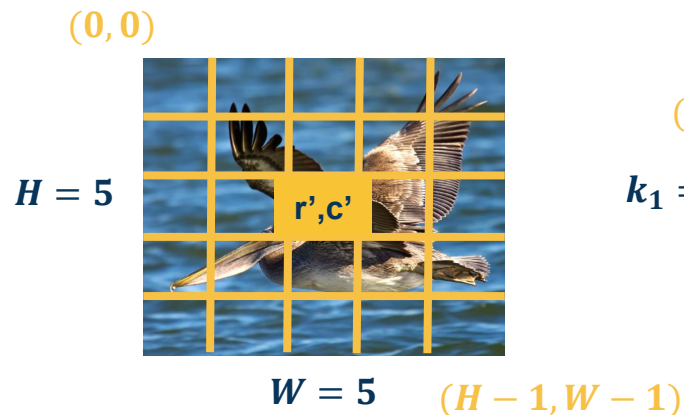
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial x}$$

Gradient for input (to pass to prior layer)

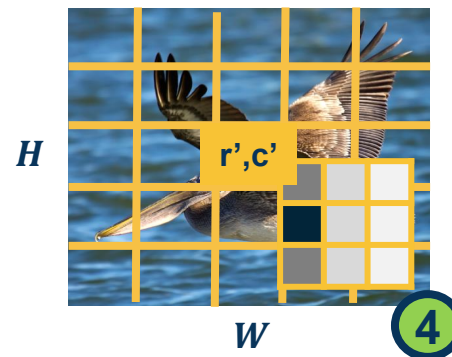
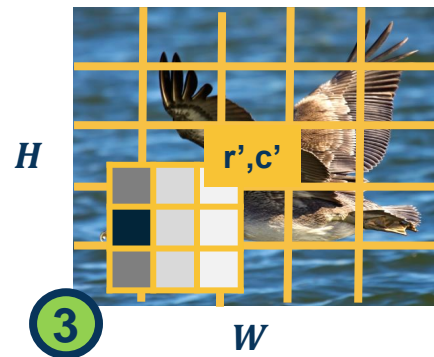
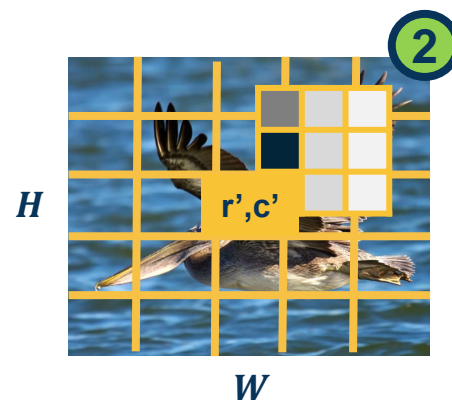
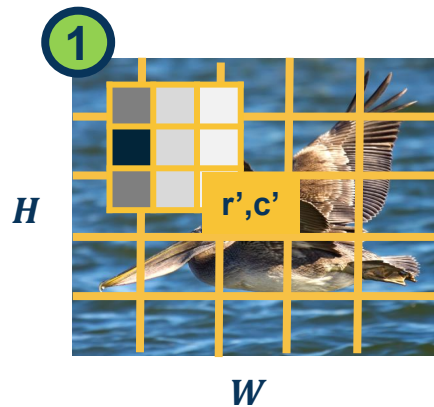
Calculate one pixel at a time  $\frac{\partial L}{\partial x(r', c')}$

What does this input pixel affect at the output?

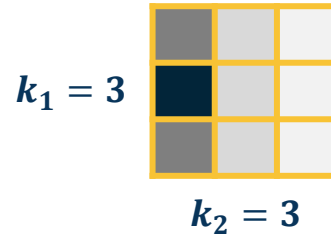
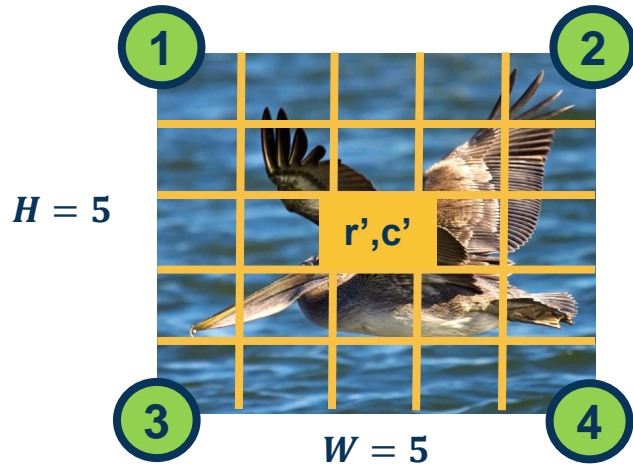
Neighborhood around it  
(where part of the kernel touches it)



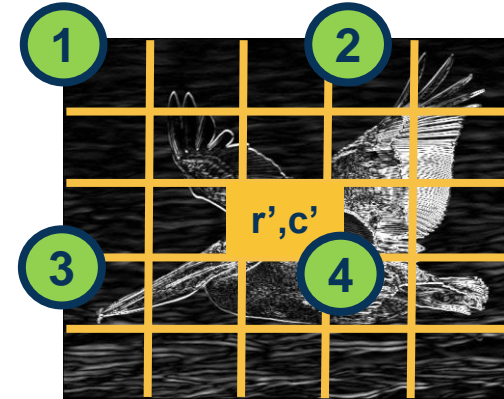
What an Input Pixel Affects at Output



## Extents of Kernel Touching the Pixel



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



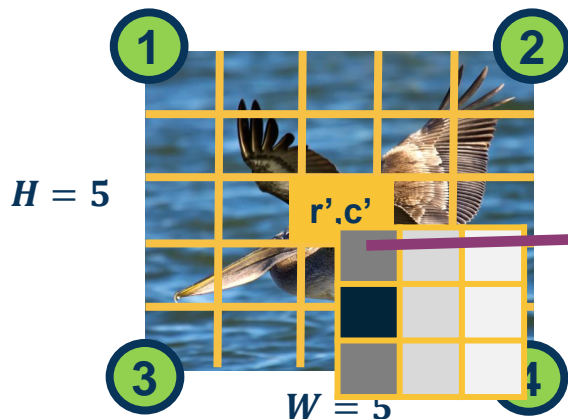
This is where the corresponding locations are for the **output**

## Extents at the Output

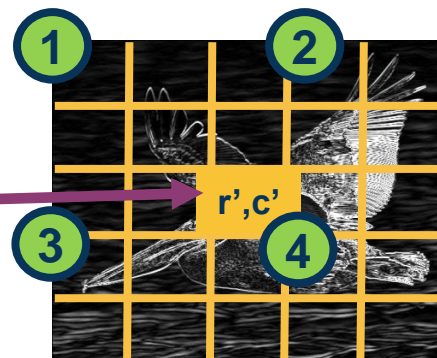
Chain rule for affected pixels (sum gradients):

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

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



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



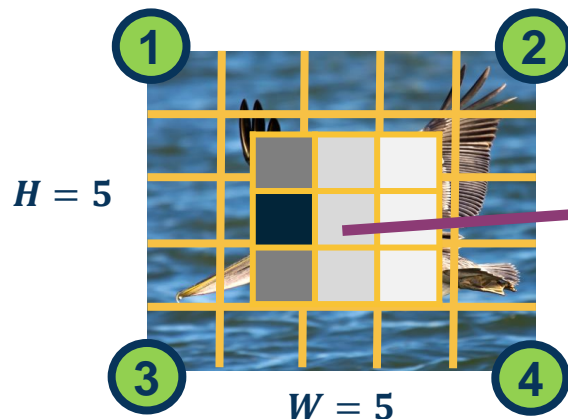
Summing Gradient Contributions

Chain rule for affected pixels (sum gradients):

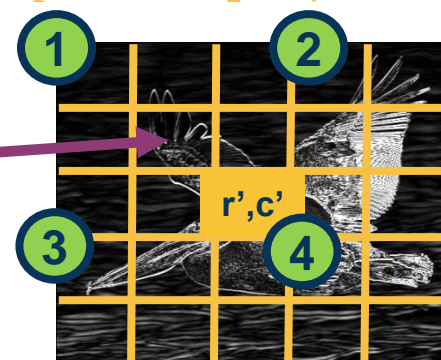
$$\frac{\partial L}{\partial x(r', c')} = \sum_{\text{Pixels } p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r', c')}$$

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

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)