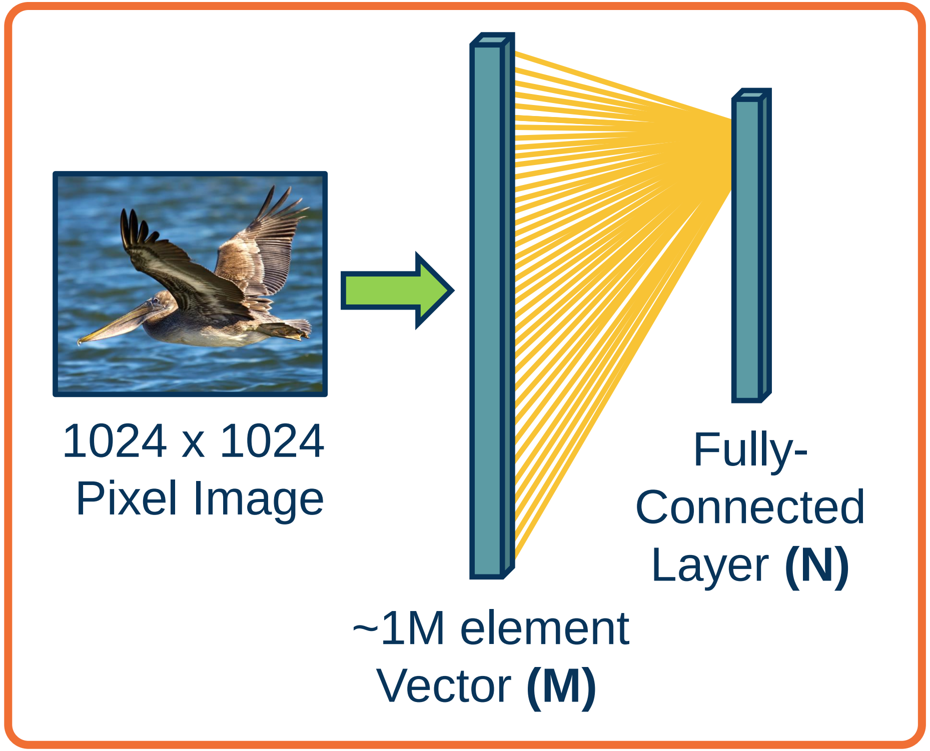
Topics:

- Convolutional Neural Networks

CS 4644-DL / 7643-A
ZSOLT KIRA

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**

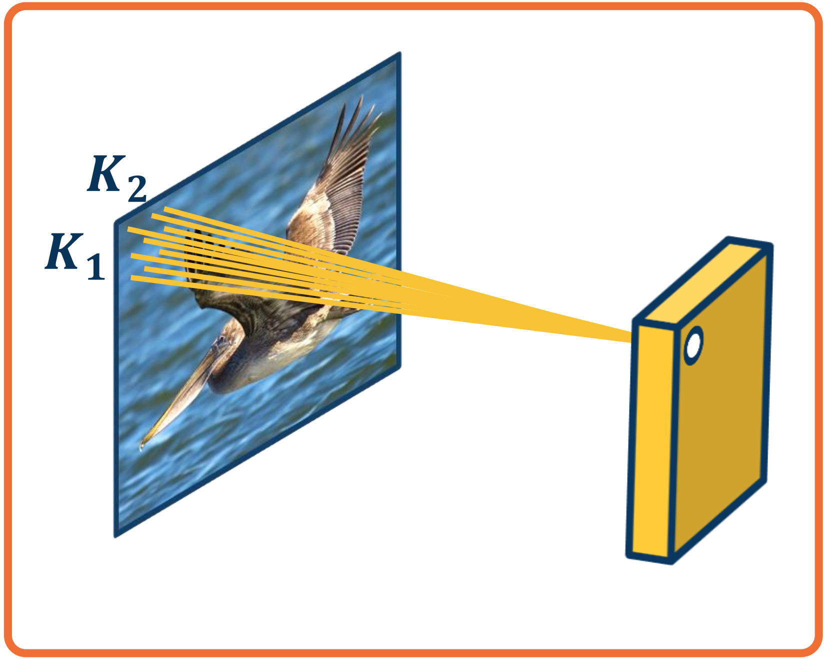
More parameters => More data needed

Is this necessary?

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)

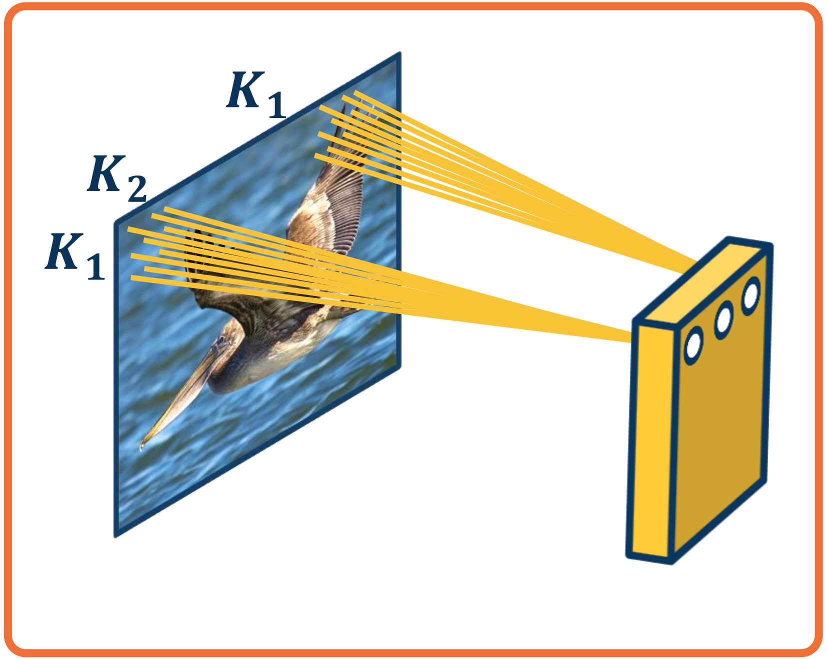
- Region from which a node receives input from is called its **receptive field**

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



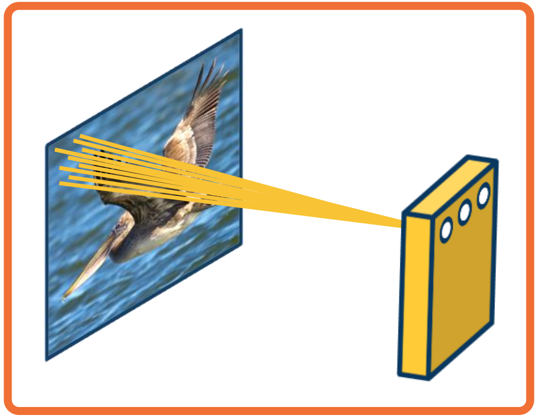
Nodes in different locations can **share** features

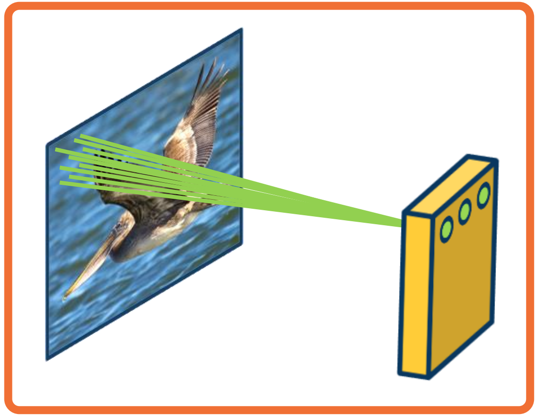
- No reason to think same feature (e.g. edge pattern) can't appear elsewhere
- Use same weights/parameters in computation graph **(shared weights)**

Advantages:

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

Idea 2: Shared Weights



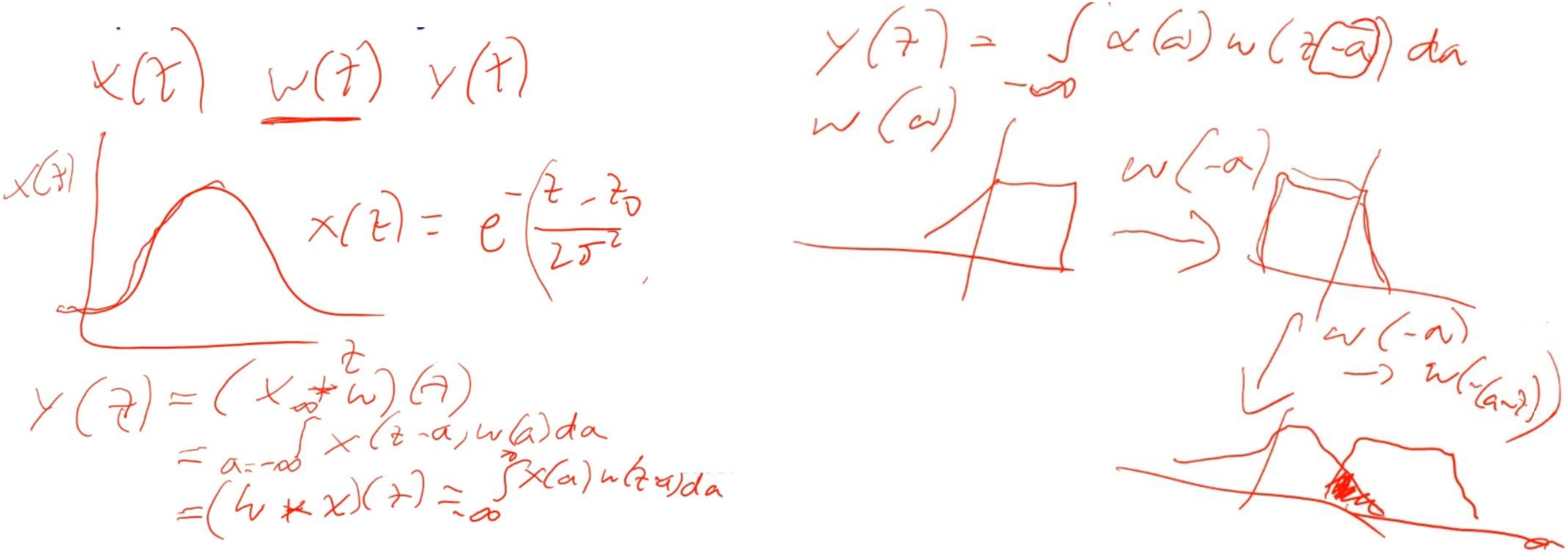


We can learn **many** such features for this one layer

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

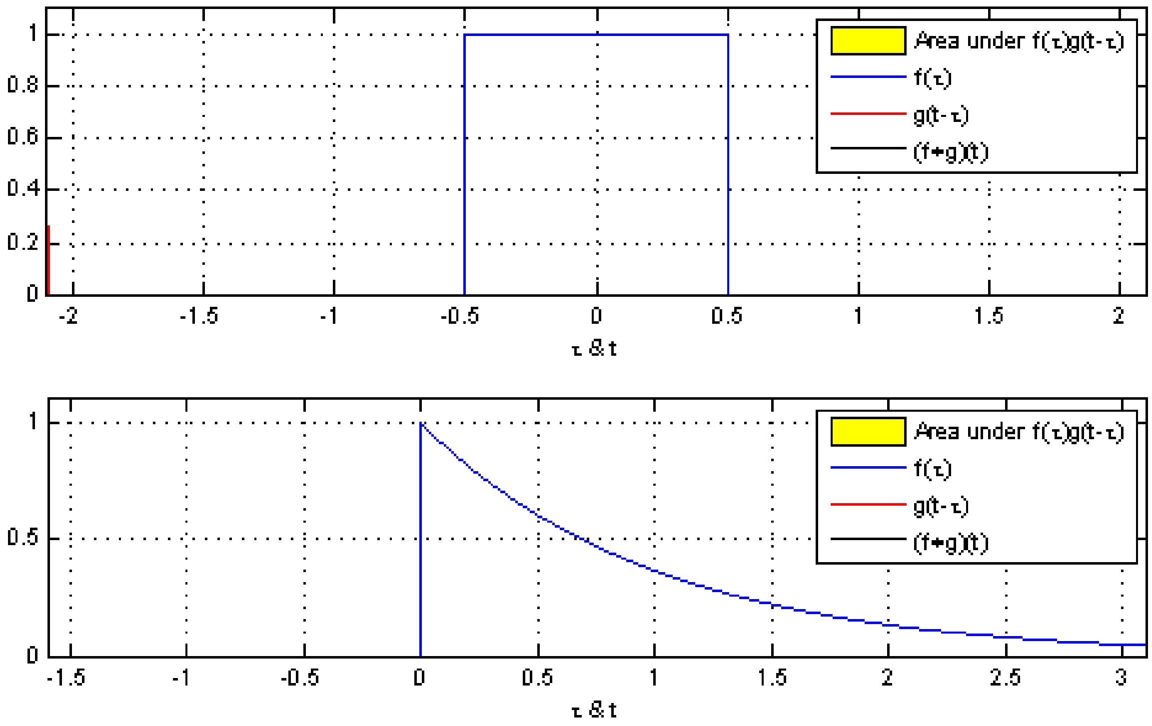
Idea 3: Learn Many Features

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!



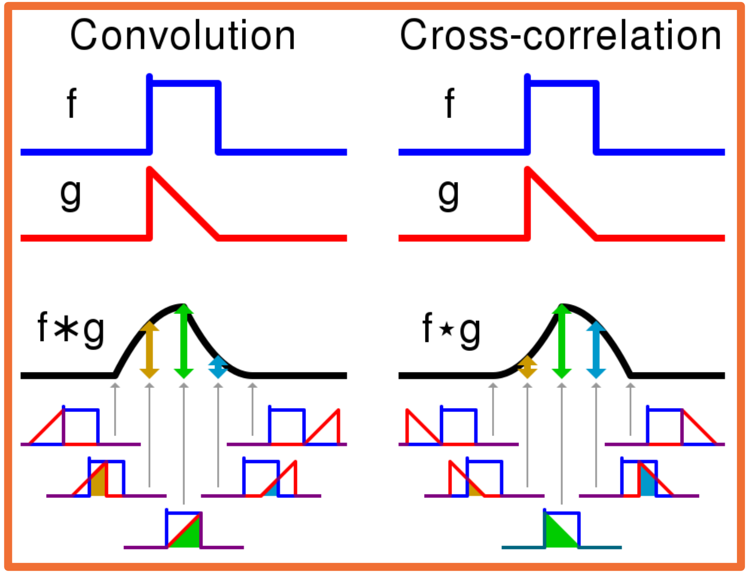
From https://en.wikipedia.org/wiki/Convolution

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

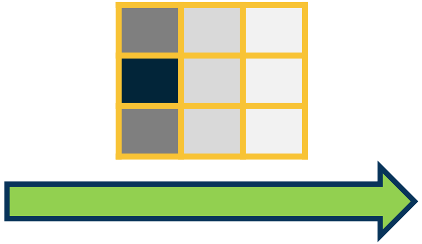
1D Convolution

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

$$y_0 = h_0 \cdot x_0$$
$$y_1 = h_1 \cdot x_0 + h_0 \cdot x_1$$
$$y_2 = h_2 \cdot x_0 + h_1 \cdot x_1 + h_0 \cdot x_2$$
$$y_3 = h_3 \cdot x_0 + h_2 \cdot x_1 + h_1 \cdot x_2 + h_0 \cdot x_3$$
$$\vdots$$

2D Convolution

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

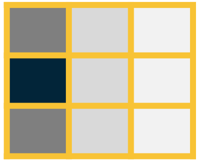


2D Convolution

Image

Kernel
(or filter)

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



Output /
filter /
feature map

2D Discrete Convolution

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

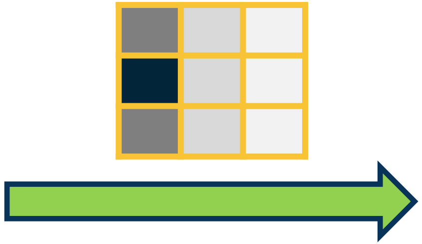
- Initialize kernel values randomly and optimize them!
- These are our parameters (plus a bias term per filter)

2D Convolution

Image

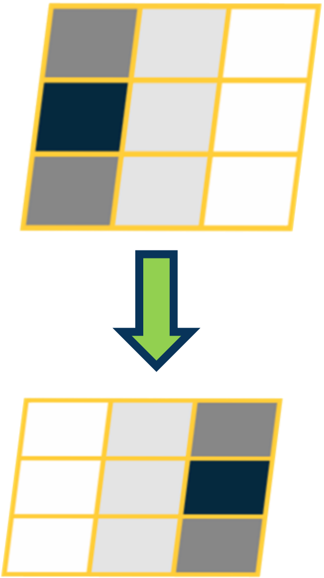
Kernel
(or filter)

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

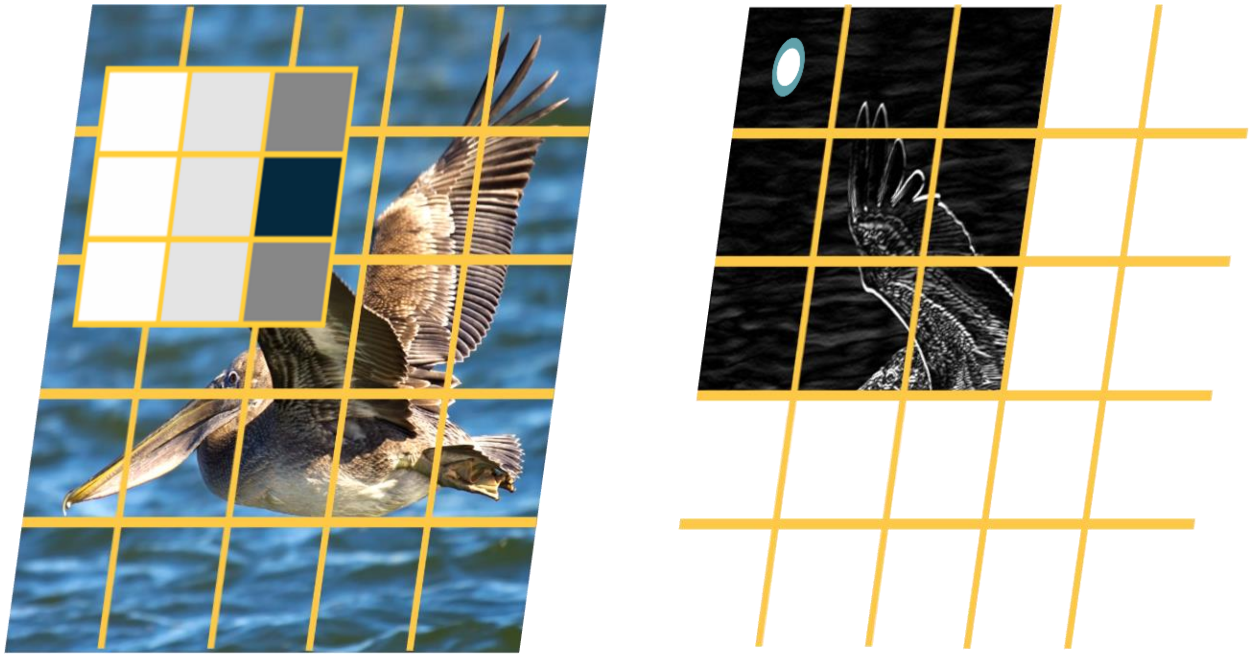


Output /
filter /
feature map

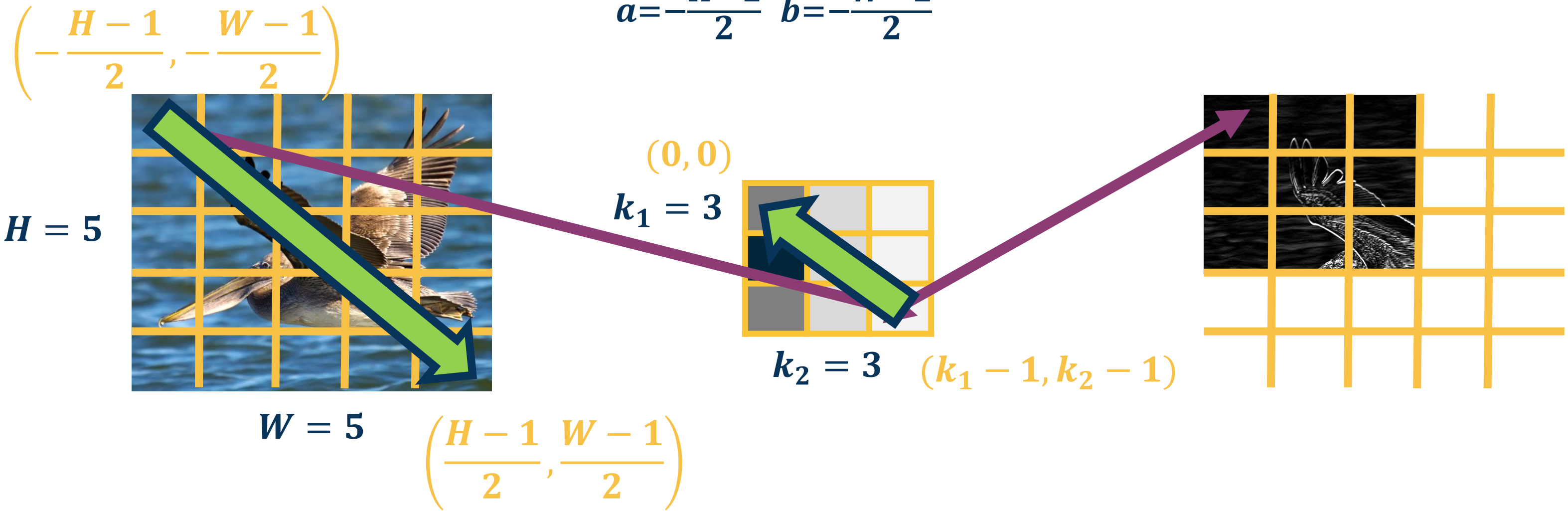
1. Flip kernel (rotate 180 degrees)



2. Stride along image



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



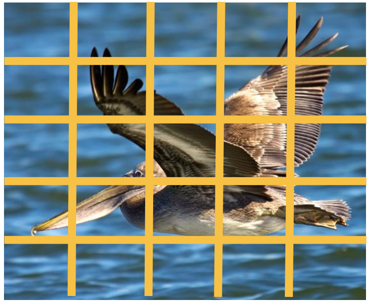
$$y(0,0) = x(-2,-2)k(2,2) + x(-2,-1)k(2,1) + x(-2,0)k(2,0) +$$
$$x(-2,1)k(2,-1) + x(-2,2)k(2,-2) + \ldots$$

Mathematics of Discrete 2D Convolution

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

$(0,0)$

$H = 5$

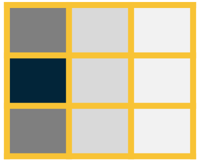


$W = 5$

$(H-1, W-1)$

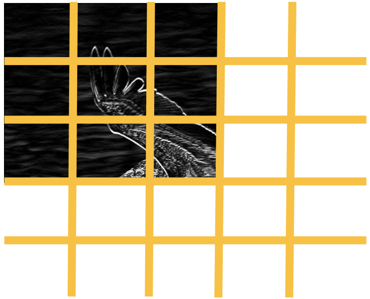
$\left(-\frac{k_1-1}{2}, -\frac{k_2-1}{2}\right)$

$k_1 = 3$



$k_2 = 3$

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



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:

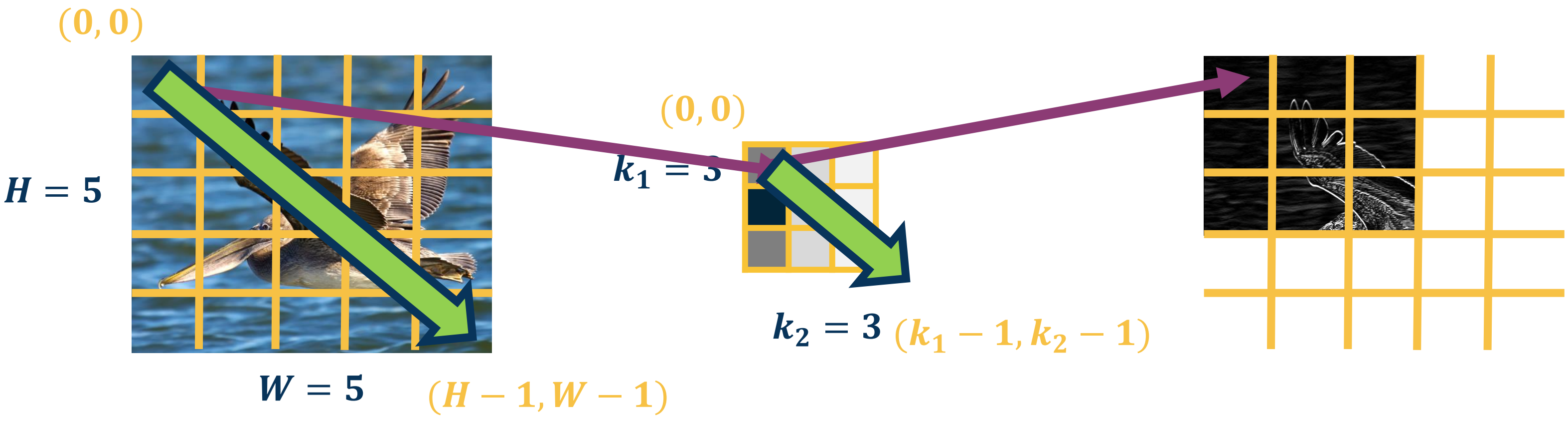
- Take the kernel, and rotate 180 degrees along center (sometimes referred to as "flip")
- Perform cross-correlation
- (Just dot-product filter with image!)

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

$$\Downarrow$$

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

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



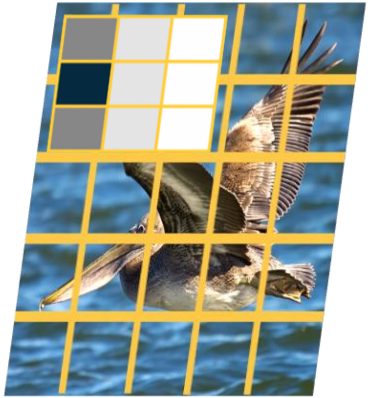
Since we will be learning these kernels, this change does not matter!

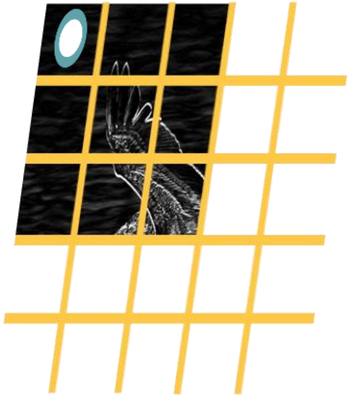
Cross-Correlation

$$X(0{:}2, 0{:}2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix} \qquad K' = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

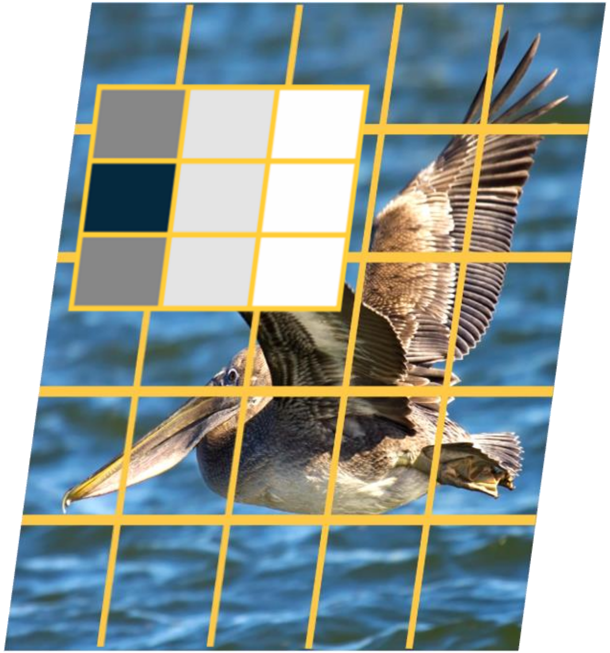
$$X(0{:}2,0{:}2) \cdot K' = 65 + \text{bias}$$

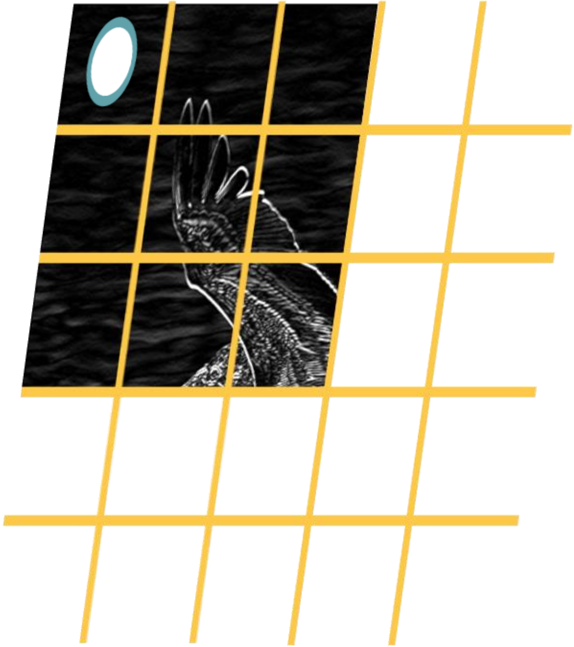
Dot product
(element-wise multiply and sum)



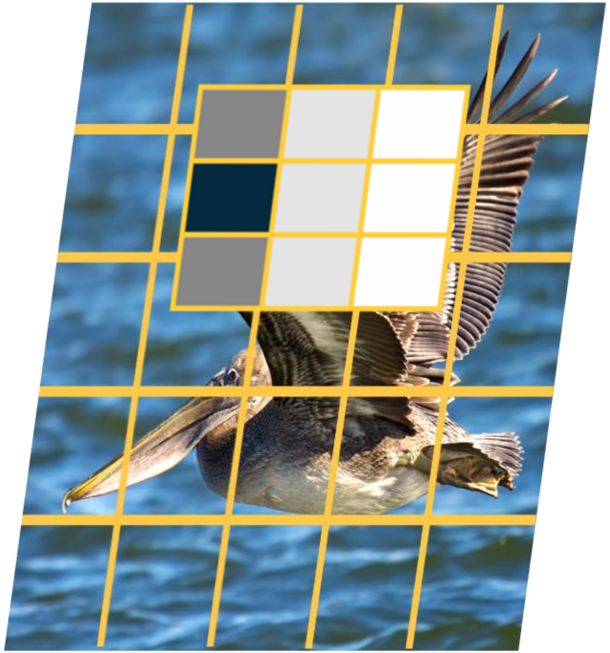


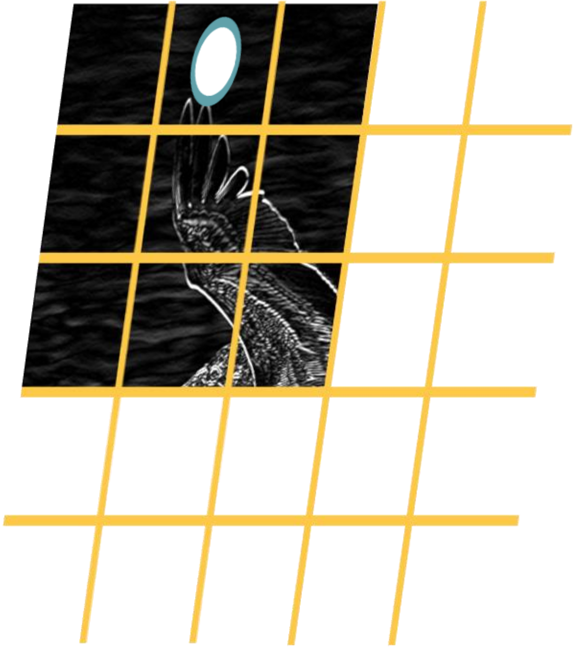
Cross-Correlation



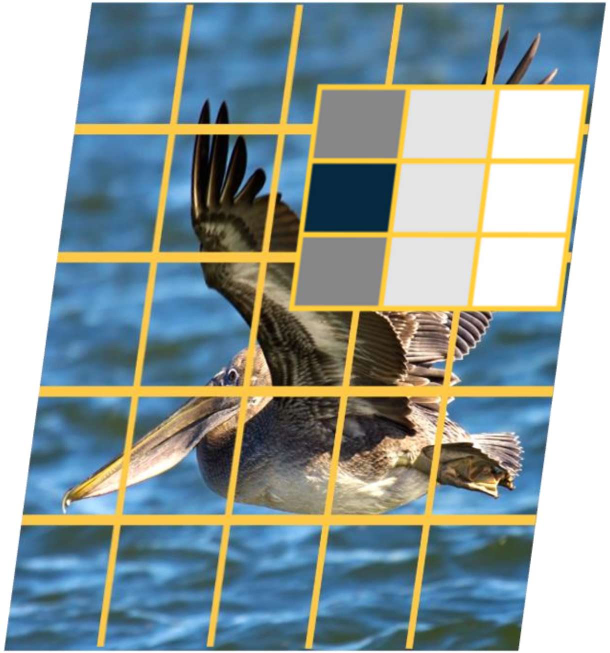


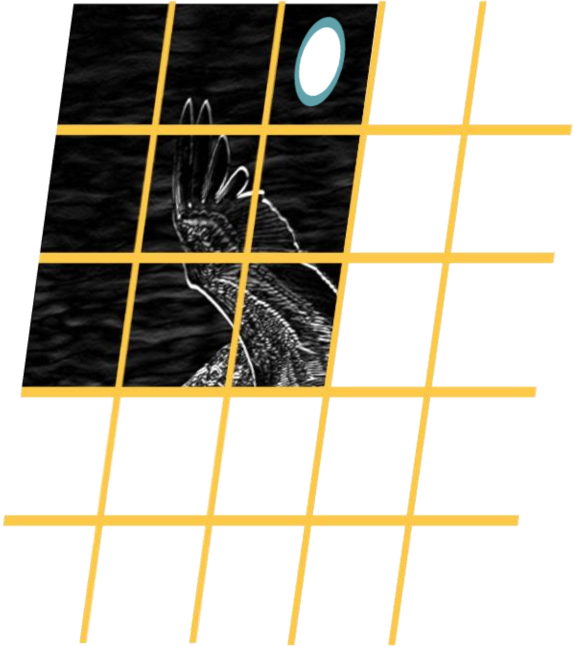
Convolution and Cross-Correlation



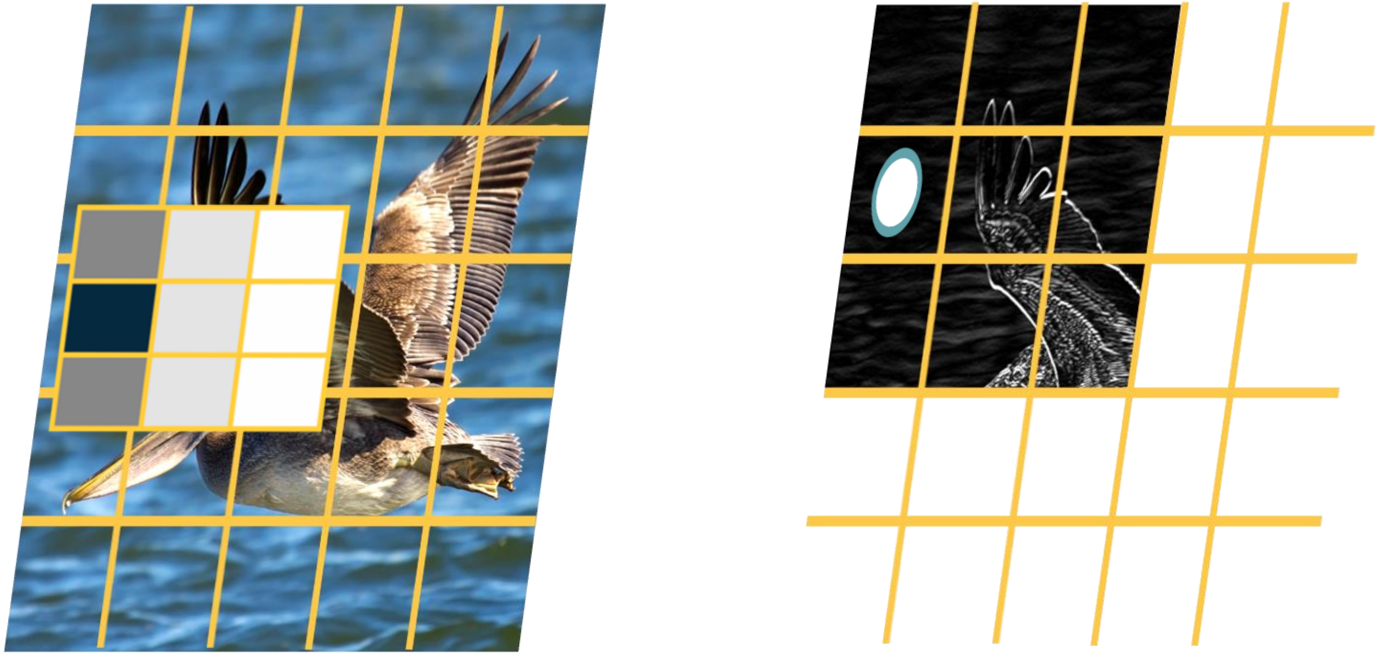


Convolution and Cross-Correlation

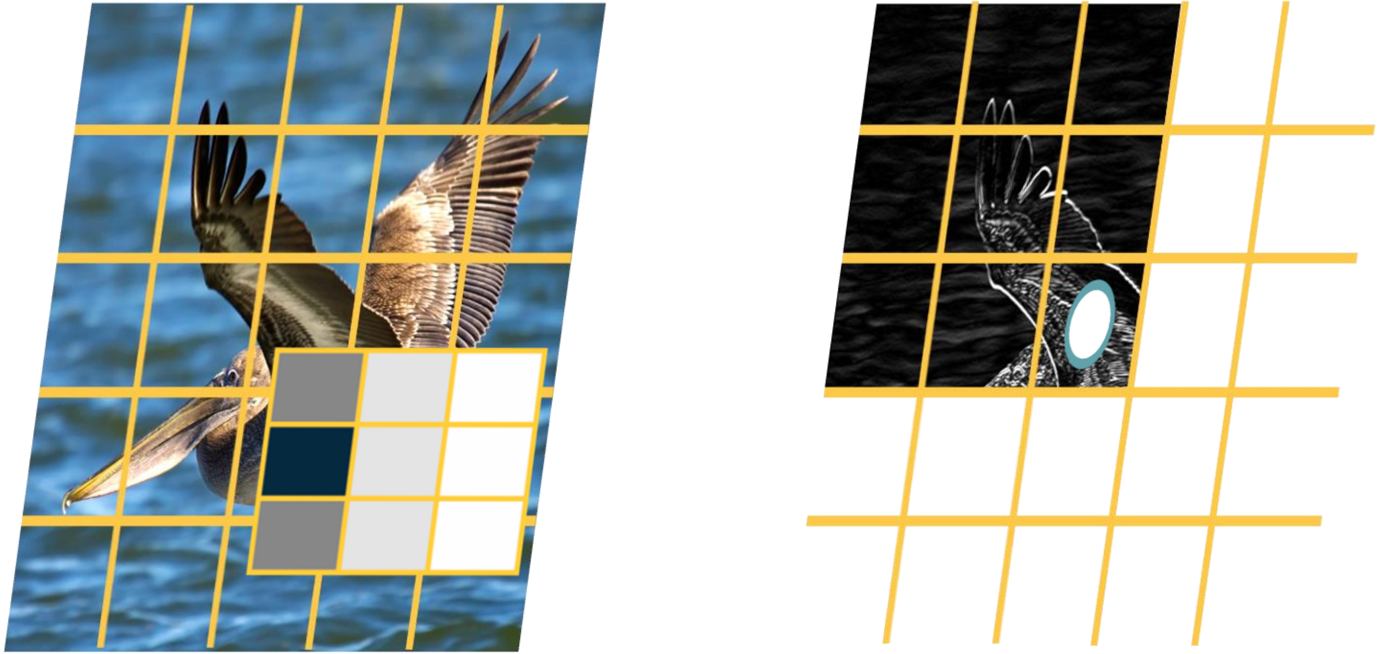




Convolution and Cross-Correlation



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

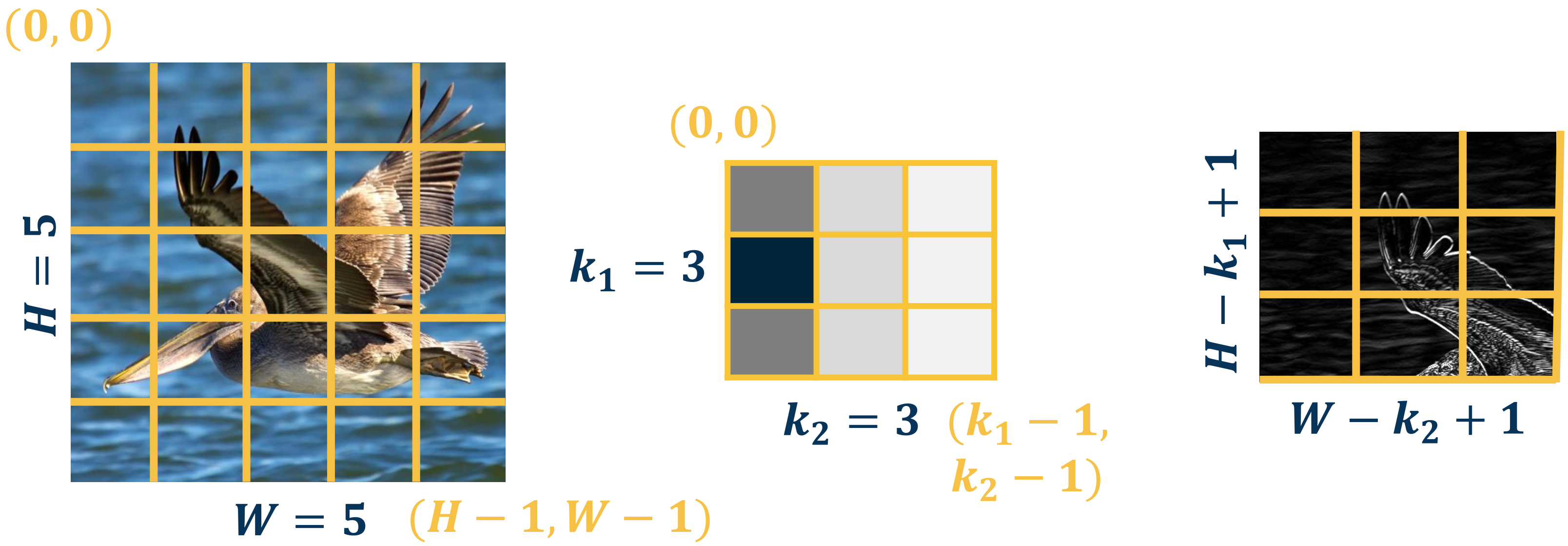
- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*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

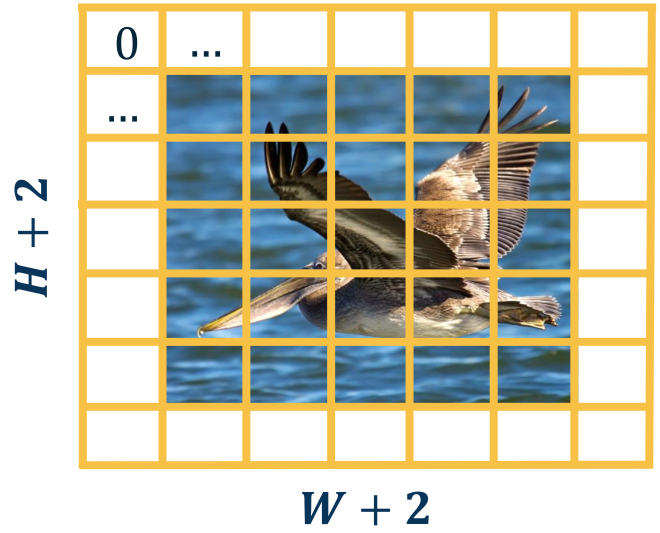
Output size of vanilla convolution operation is $(H - k_1 + 1) \times (W - k_2 + 1)$

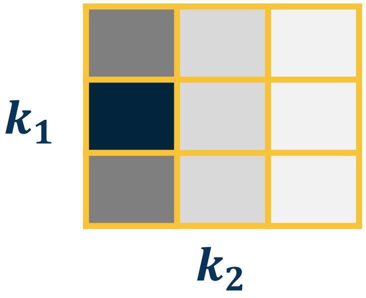
- This is called a **"valid" convolution** and only applies kernel within image

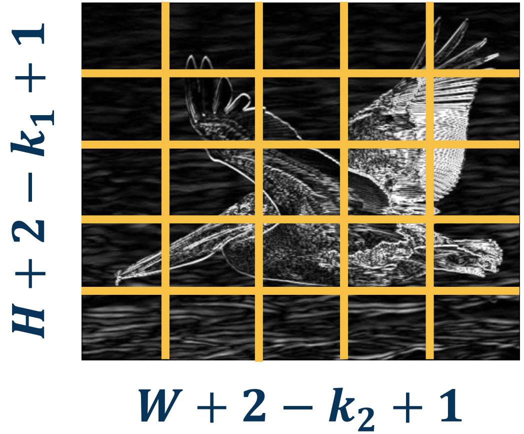


We can **pad the images** to make the output the same size:

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







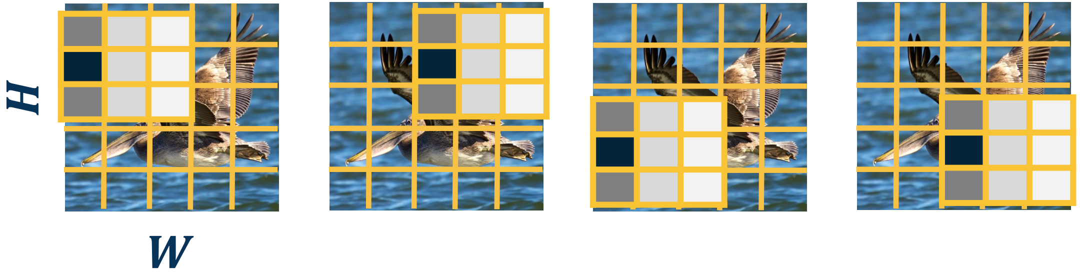
Adding Padding

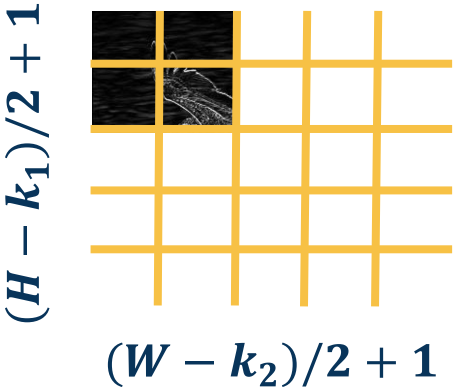
Georgia Tech

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)





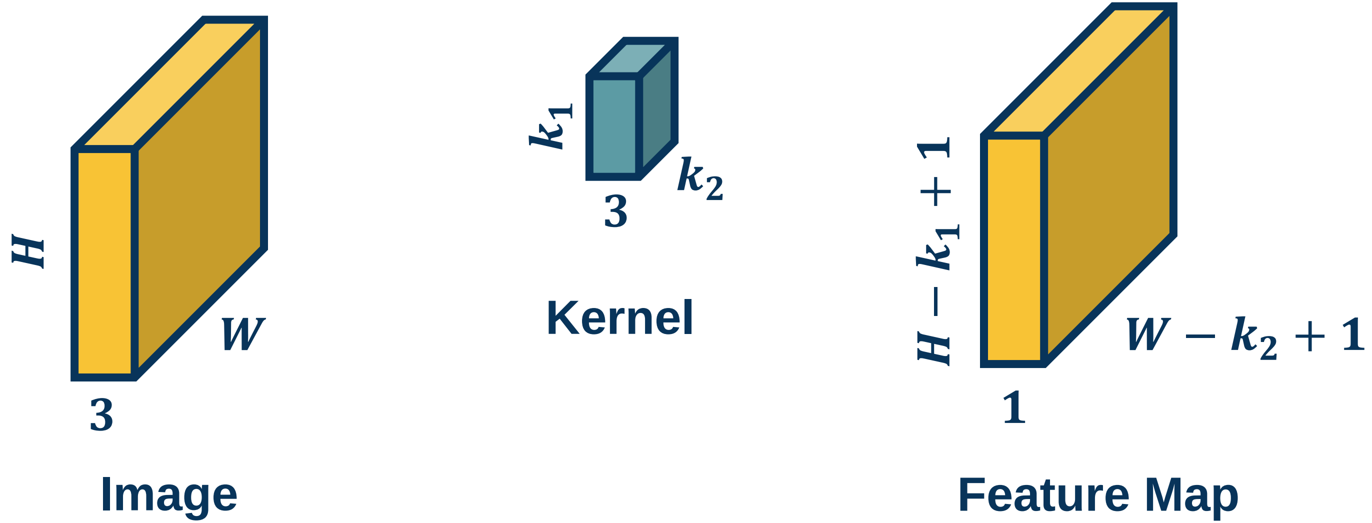
Stride

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)

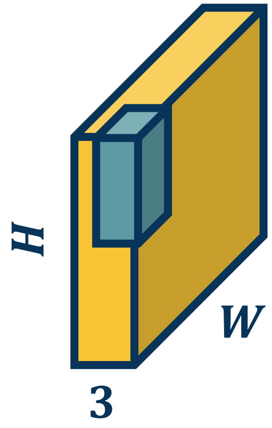
- In such cases, we have **3-channel kernels**!



Multi-Channel Inputs

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

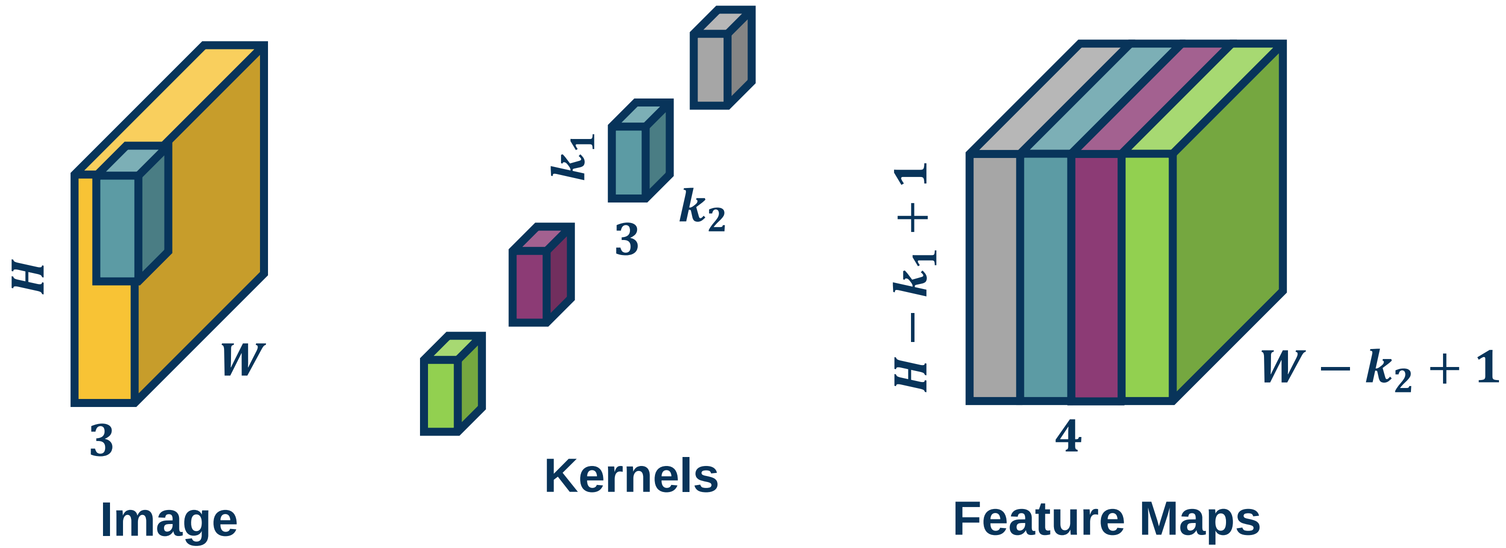
Similar to before, we perform **element-wise multiplication** between kernel and image patch, summing them up **(dot product)**

- Except with $k_1 * k_2 * 3$ values

We can have **multiple kernels per layer**

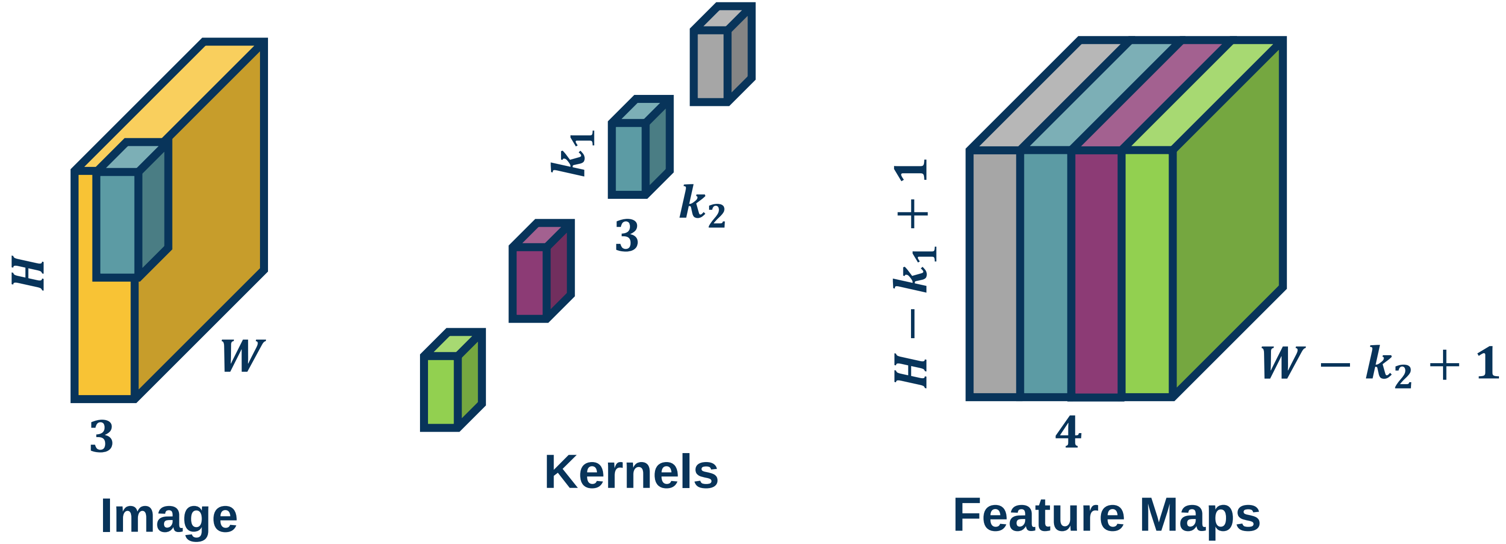
- We stack the feature maps together at the output

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



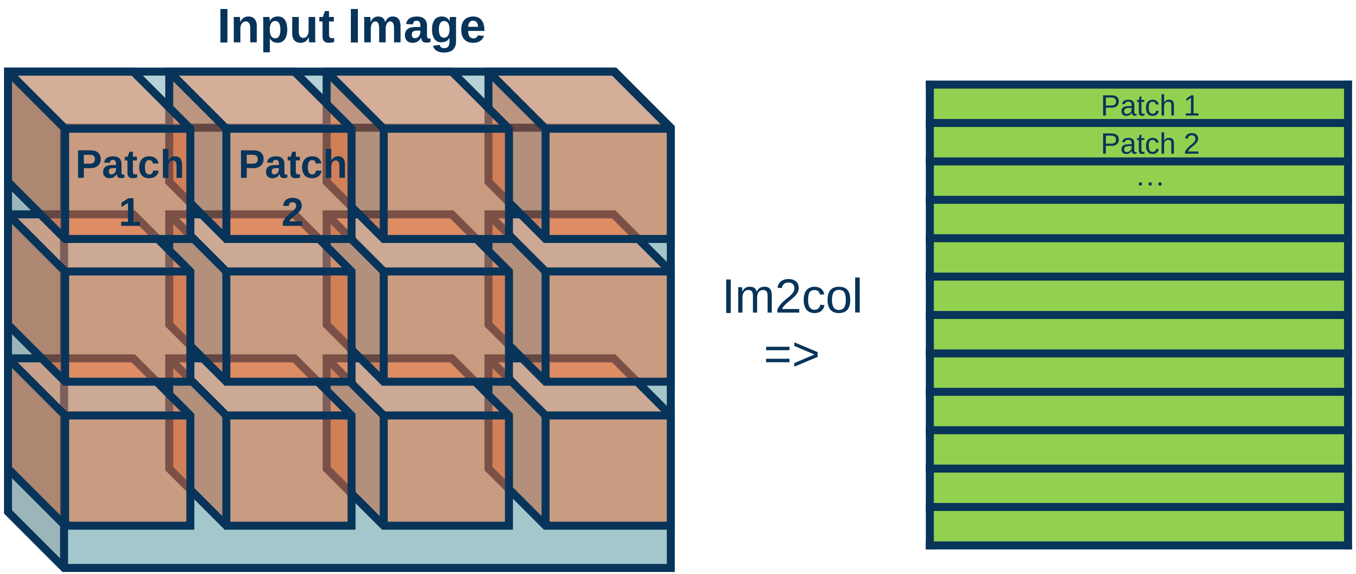
Number of parameters with N filters is: $\boldsymbol{N * (k_1 * k_2 * 3 + 1)}$

- Example:
  $\boldsymbol{k_1 = 3, k_2 = 3,\ N = 4\ input\ channels = 3}$, then $\boldsymbol{(3 * 3 * 3 + 1) * 4 = 112}$



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

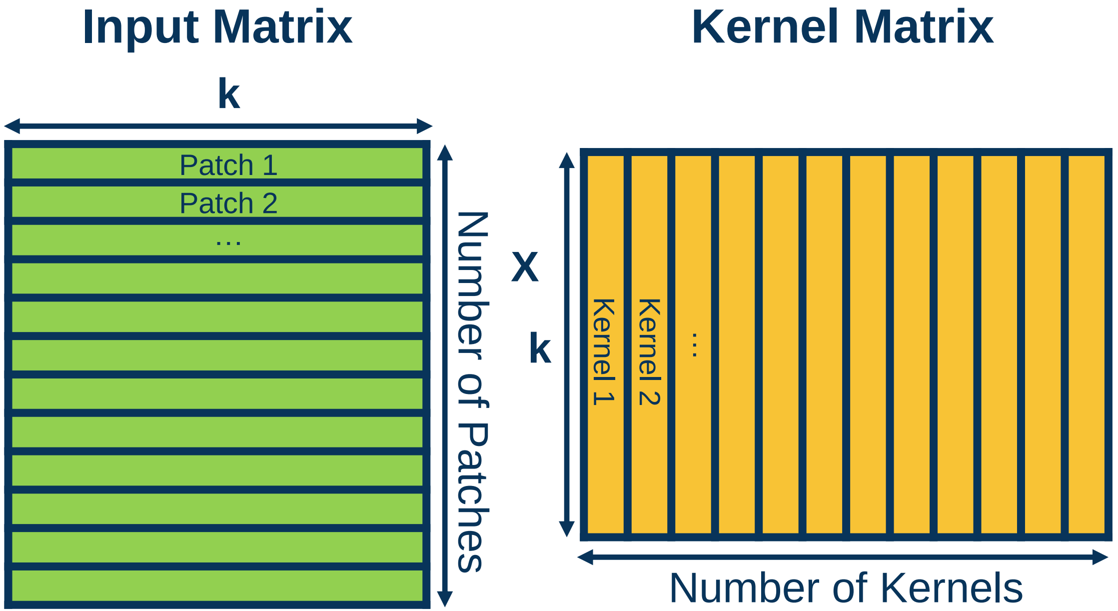
- **Step 1:** Lay out image patches in vector form (note can overlap!)



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

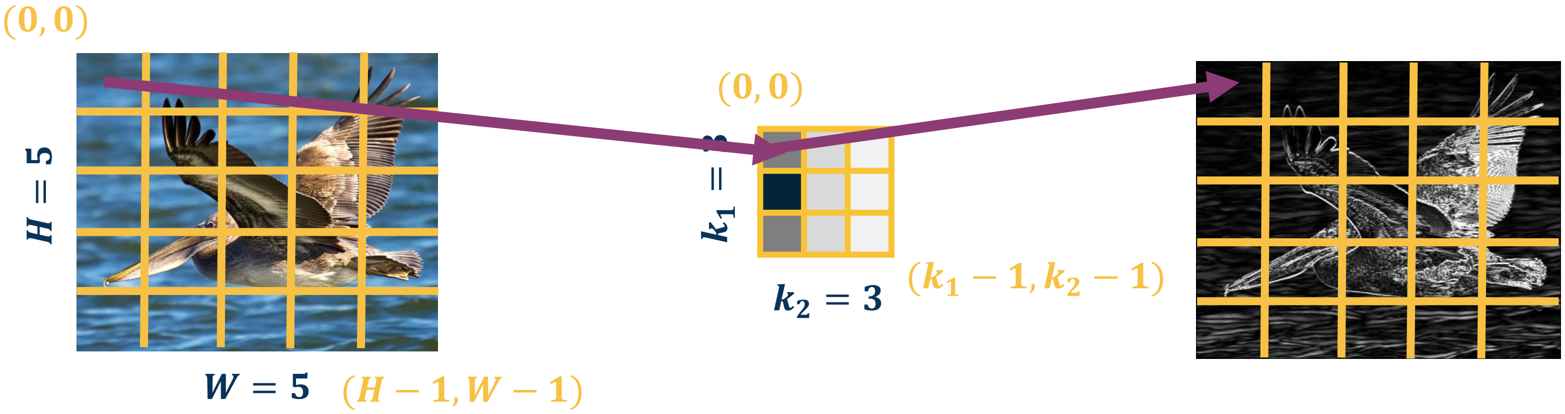
It is instructive to calculate **the backwards pass** of a convolution layer

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

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

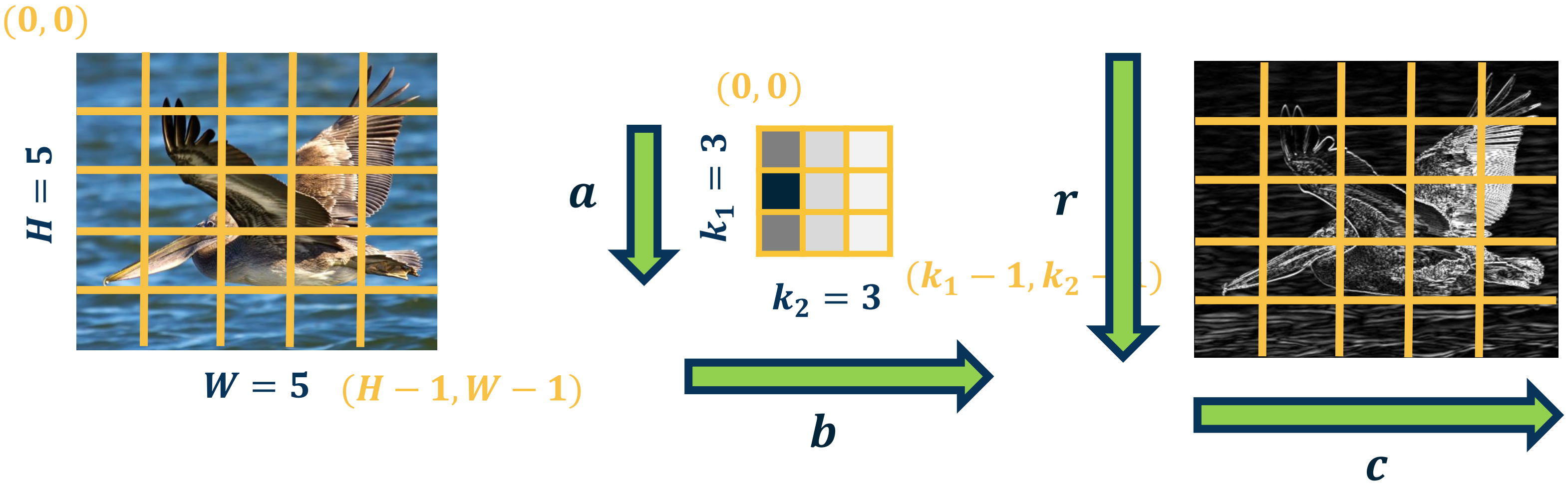
$$K' = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

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



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

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

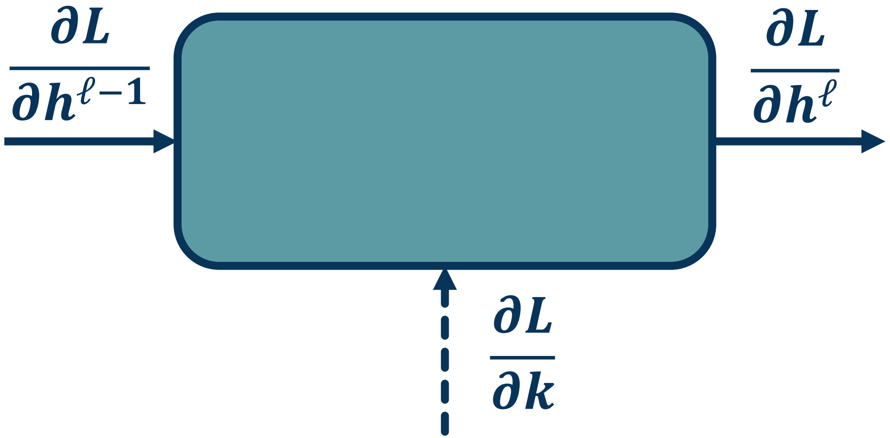
$$|y| = H \times W$$

$\frac{\partial L}{\partial y}$? Assume size $H \times W$ (add padding, change convention a bit for convenience)

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

Gradient Terms and Notation

Georgia Tech



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

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)

Gradient for Convolution Layer

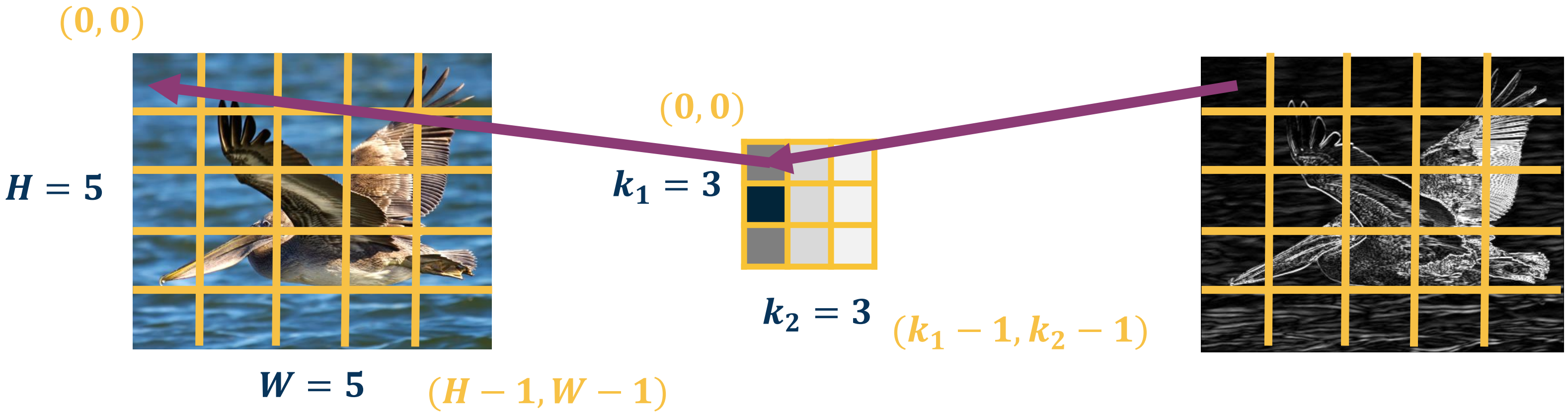
$$\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial k}$$

Gradient for weight update

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

What does this weight affect at the output?

Everything!



What a Kernel Pixel Affects at Output

Chain Rule over all Output Pixels

Need to incorporate all upstream gradients:

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

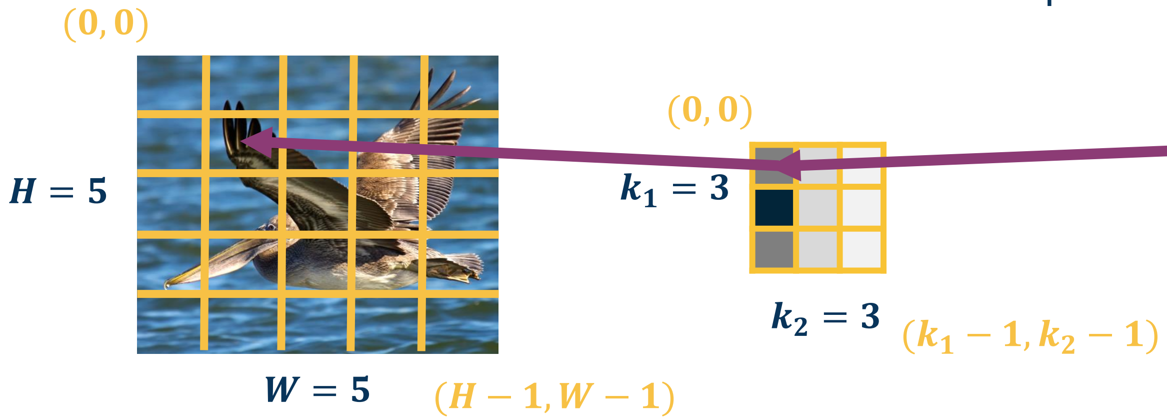
Chain Rule:

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

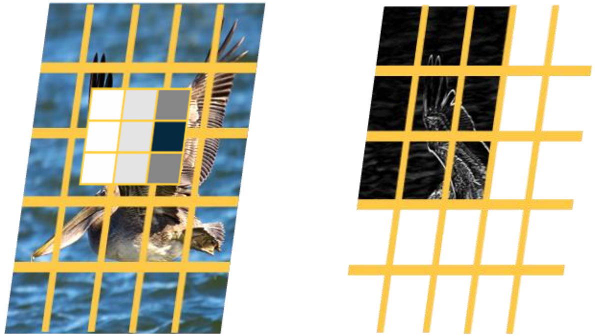
Sum over all output pixels

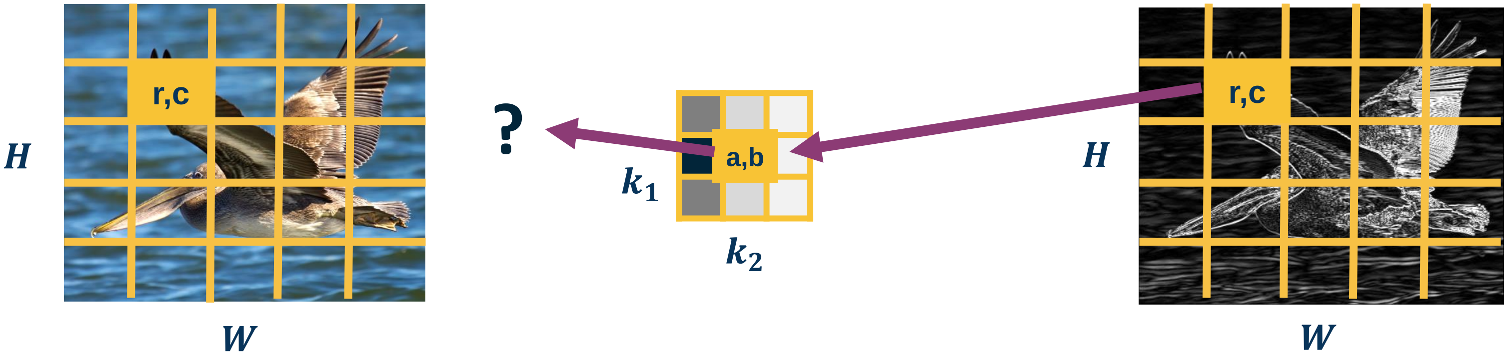
Upstream gradient (known)

We will compute



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





Chain Rule over all Output Pixels

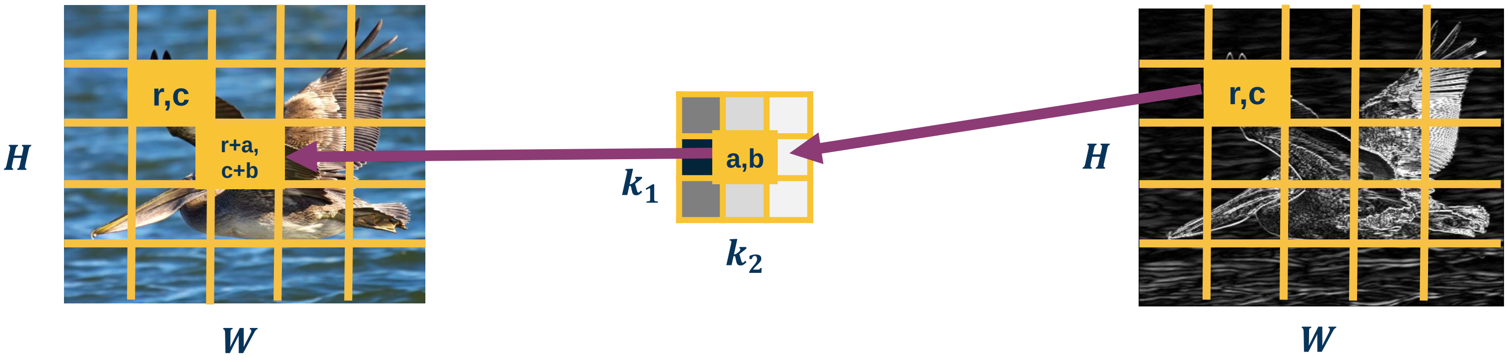
$$\frac{\partial y(r,c)}{\partial k(a,b)} = x(r+a, c+b)$$

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

Does this look familiar?

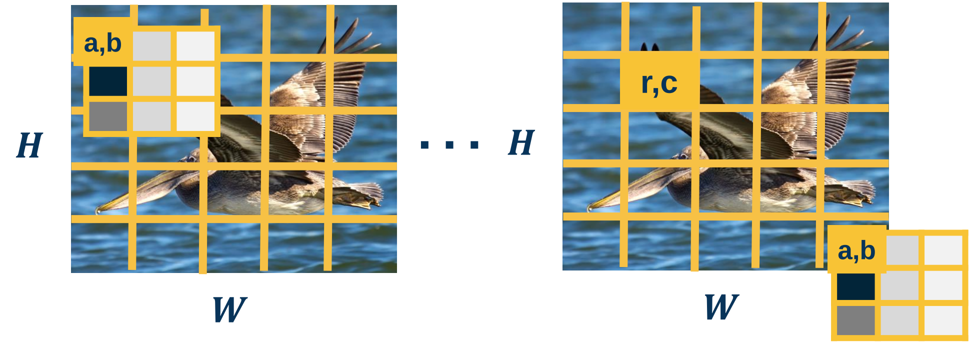
Cross-correlation between upstream gradient and input!

(until $k_1 \times k_2$ output)

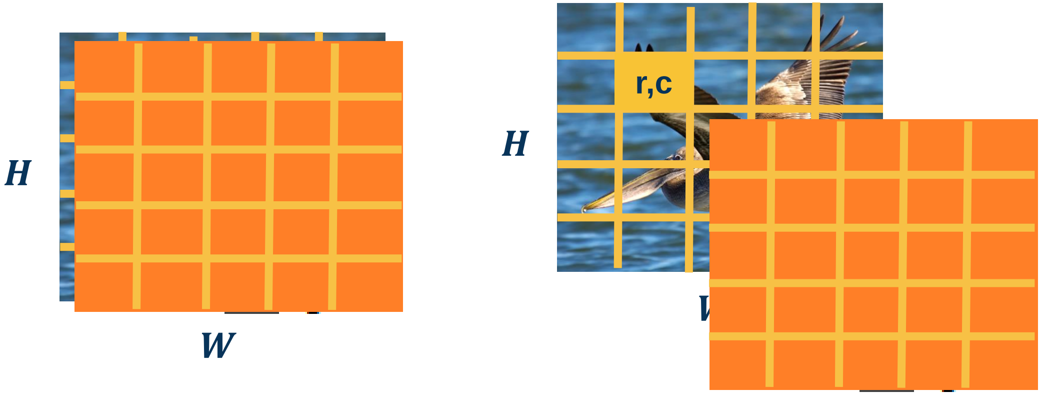


Gradients and Cross-Correlation

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

What an Input Pixel Affects at Output

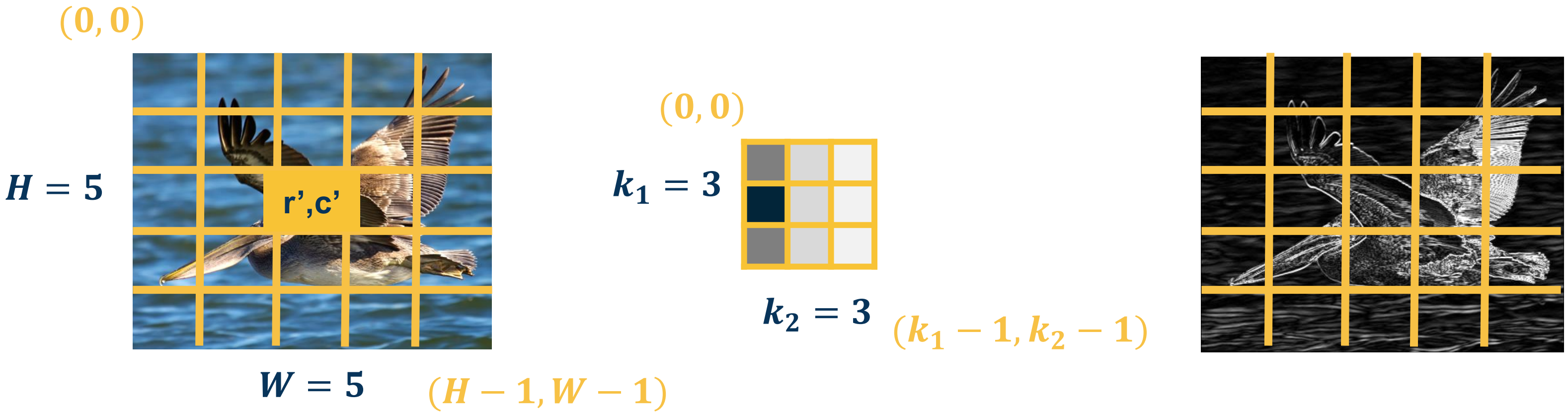
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$$

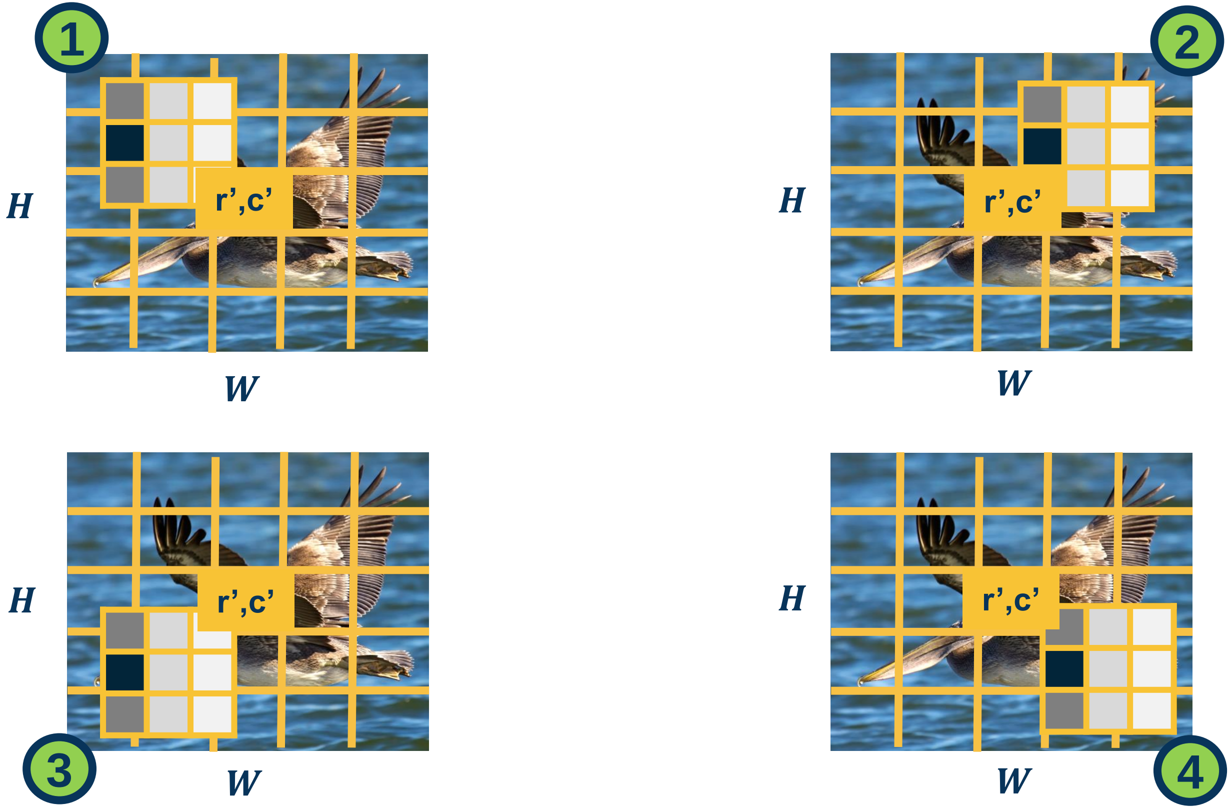
Gradient for input (to pass to prior layer)

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

What does this input pixel affect at the output?

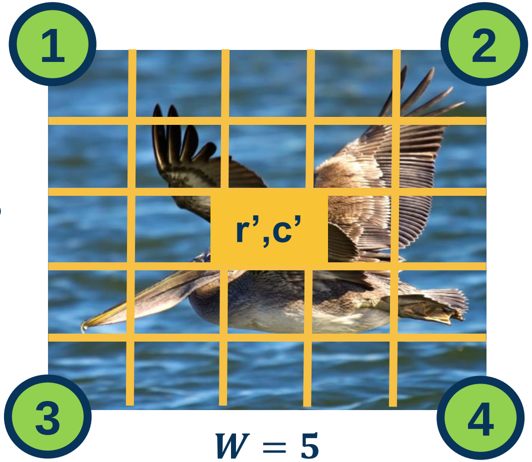
Neighborhood around it (where part of the kernel touches it)





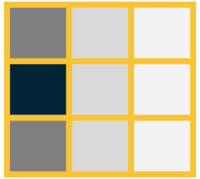
Extents of Kernel Touching the Pixel

$H = 5$



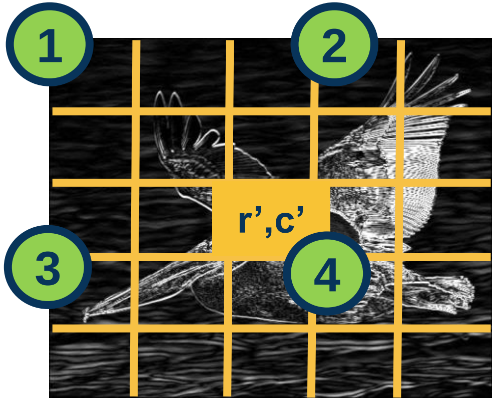
$W = 5$

$k_1 = 3$



$k_2 = 3$

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



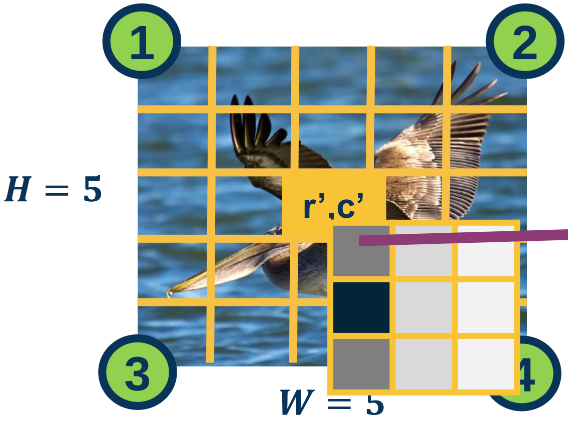
This is where the corresponding locations are for the **output**

Extents at the Output

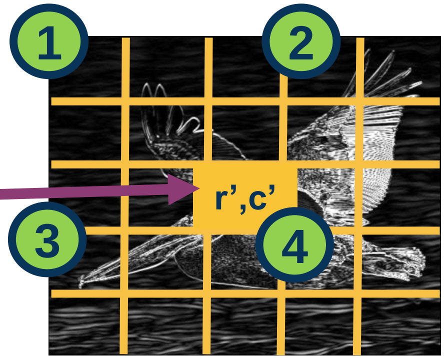
Chain rule for affected pixels (sum gradients):

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

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



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



Summing Gradient Contributions

Summing Gradient Contributions

Chain rule for affected pixels (sum gradients):

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

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

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

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

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

Plug in what we actually wanted :

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

What is $\dfrac{\partial \boldsymbol{y}(\boldsymbol{r}' - \boldsymbol{a}, \boldsymbol{c}' - \boldsymbol{b})}{\partial \boldsymbol{x}(\boldsymbol{r}', \boldsymbol{c}')} = \mathbf{k}(\boldsymbol{a}, \boldsymbol{b})$

(we want term with $\boldsymbol{x}(\boldsymbol{r}', \boldsymbol{c}')$ in it;
this happens when $\mathbf{a} = \mathbf{a}'$ and $\mathbf{b} = \mathbf{b}'$)

Plugging in to earlier equation:

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

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

Does this look familiar?

Convolution between upstream gradient and kernel!

(can implement by flipping kernel and cross- correlation)

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

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)

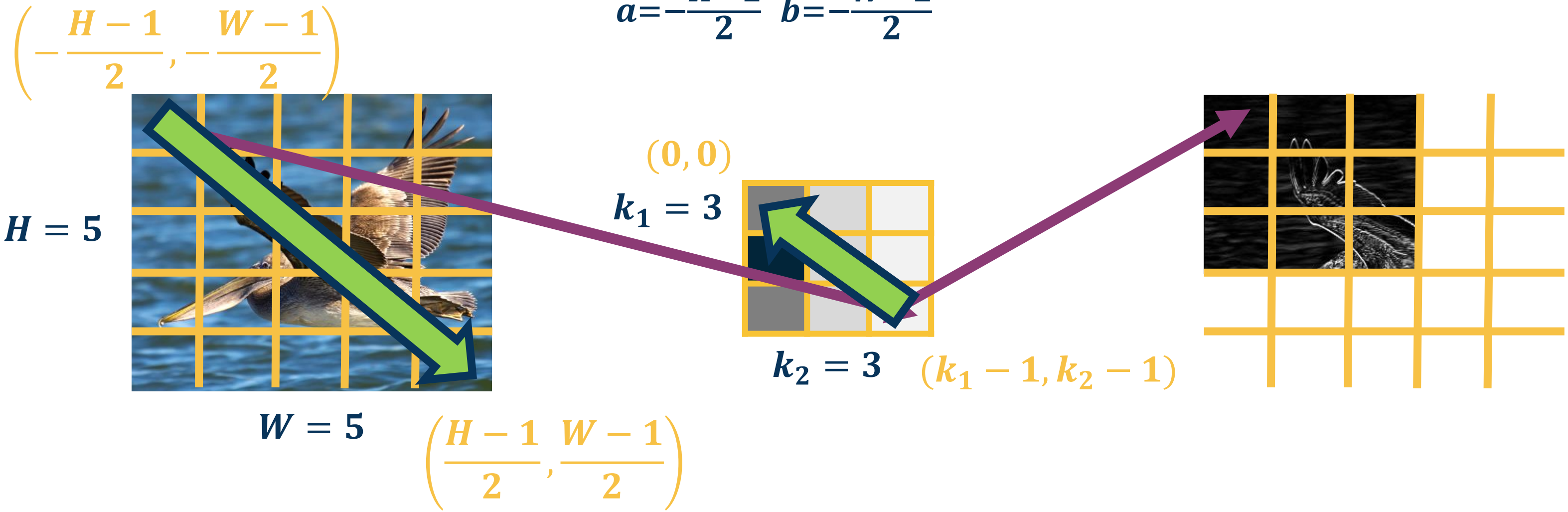
Summary

Topics:

- Convolutional Neural Networks

CS 4644-DL / 7643-A
ZSOLT KIRA

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

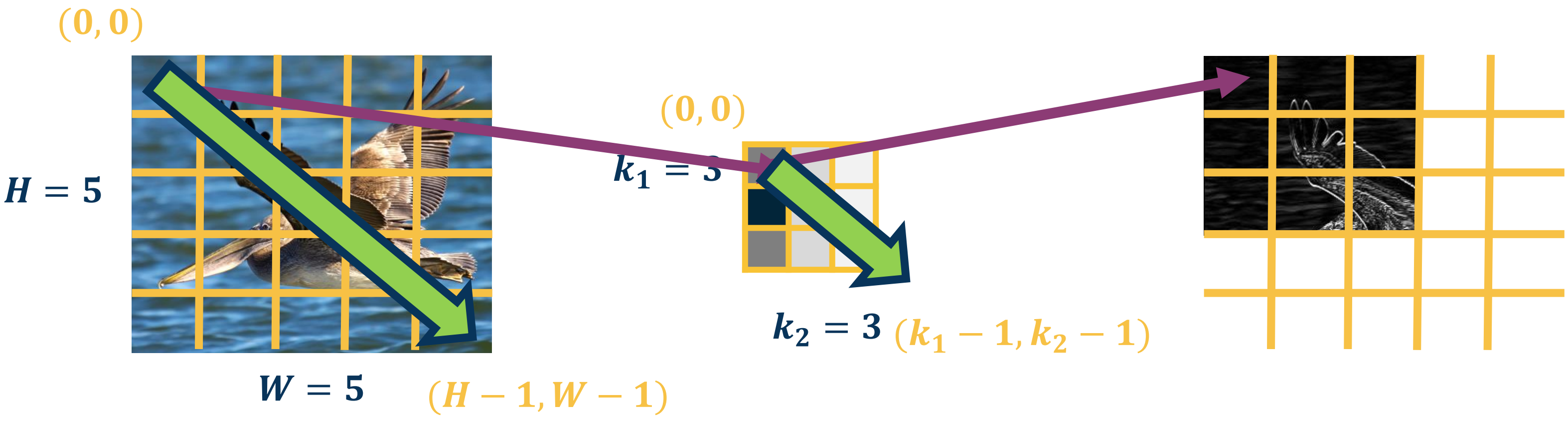


$$y(0,0) = x(-2,-2)k(2,2) + x(-2,-1)k(2,1) + x(-2,0)k(2,0) + x(-2,1)k(2,-1) + x(-2,2)k(2,-2) + \ldots$$

Mathematics of Discrete 2D Convolution

Georgia Tech

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



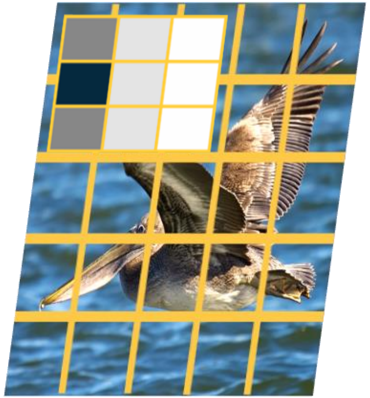
Since we will be learning these kernels, this change does not matter!

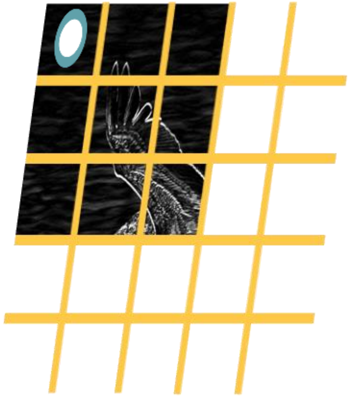
Cross-Correlation

$$X(0:2,0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix} \qquad K' = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

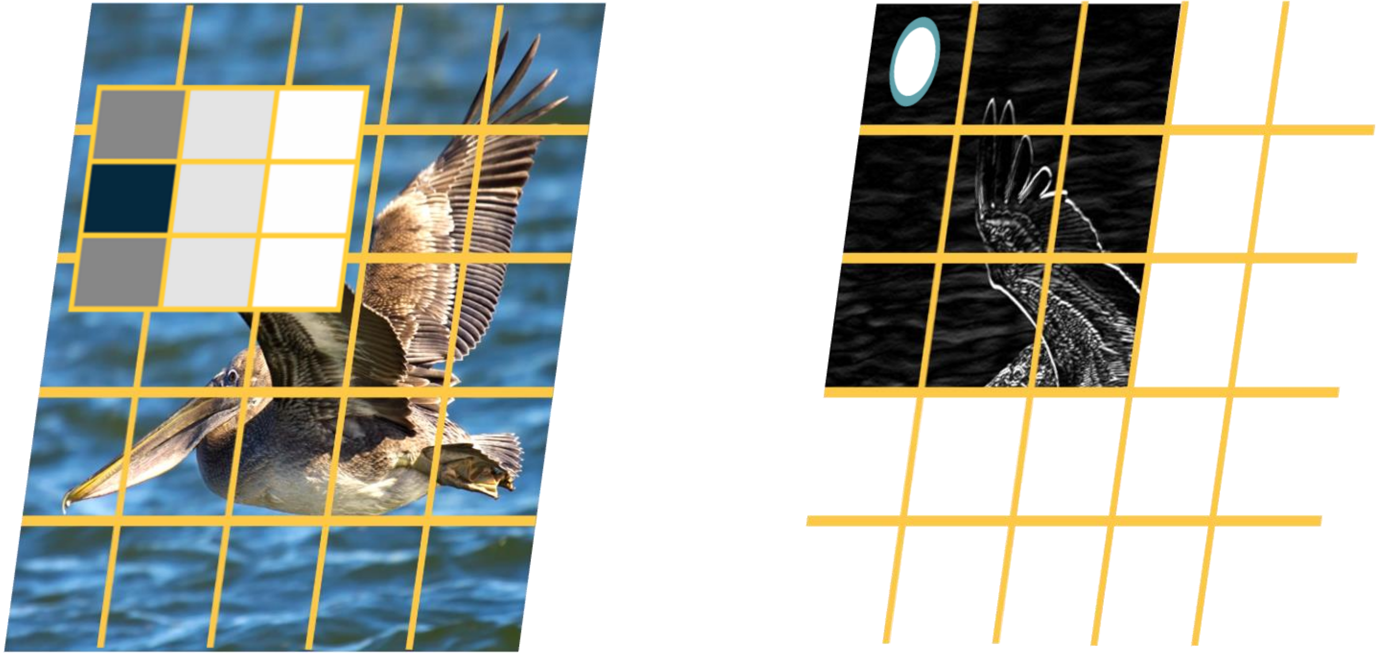
$$\Rightarrow \quad X(0:2,0:2) \cdot K' = 65 + \text{bias}$$

Dot product
(element-wise multiply and sum)

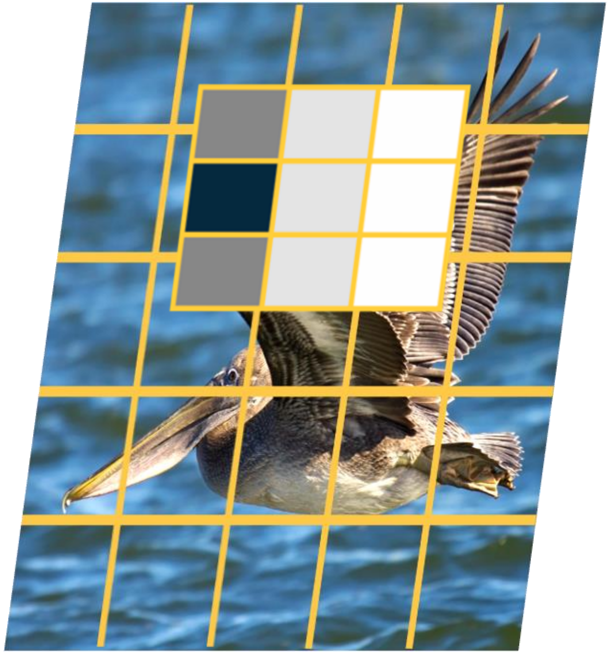


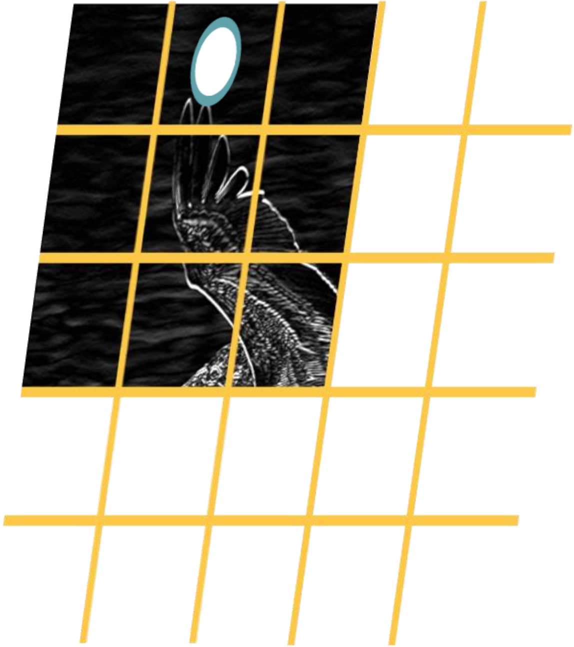


Cross-Correlation

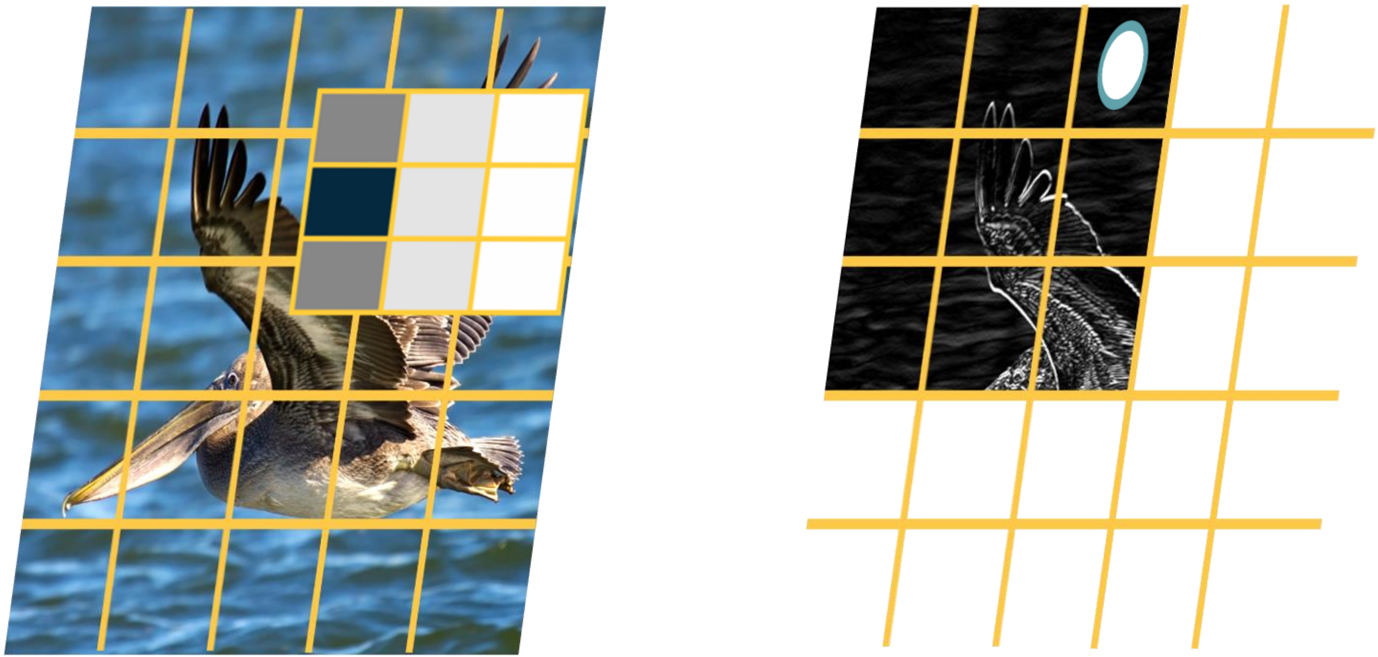


Convolution and Cross-Correlation

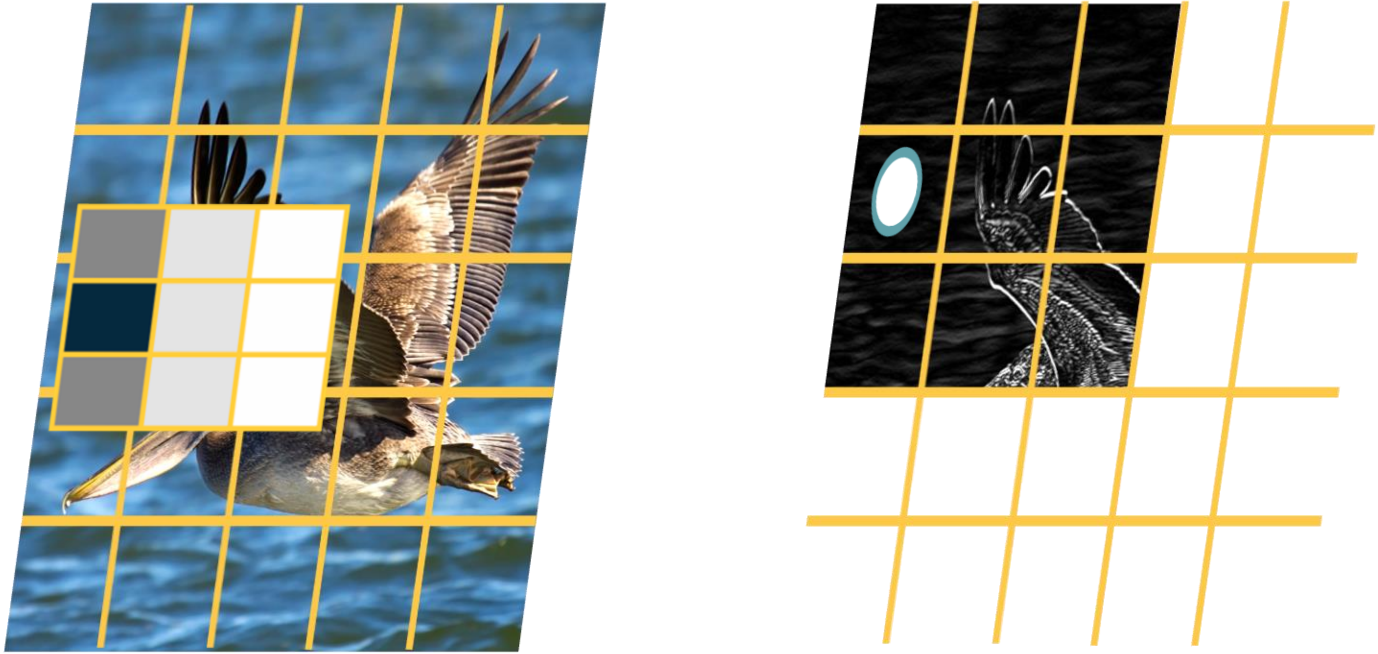




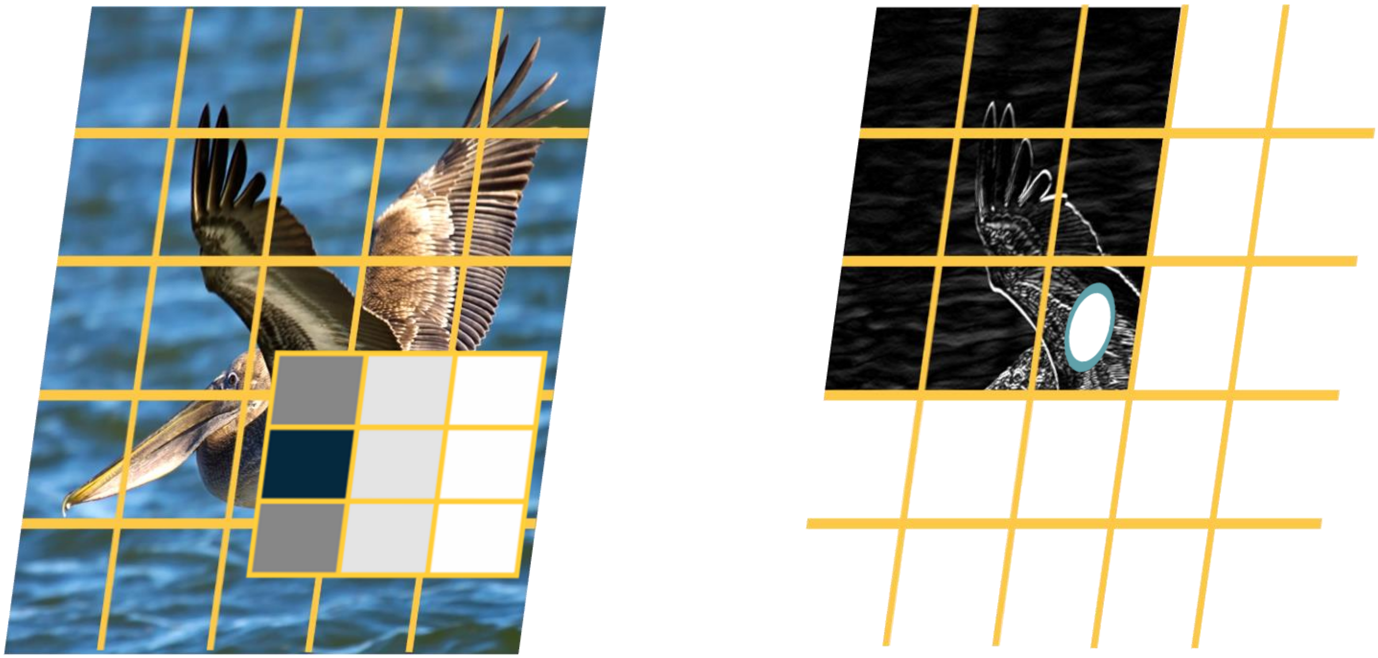
Convolution and Cross-Correlation



Convolution and Cross-Correlation



Convolution and Cross-Correlation



Convolution and Cross-Correlation

Why Bother with Convolutions?

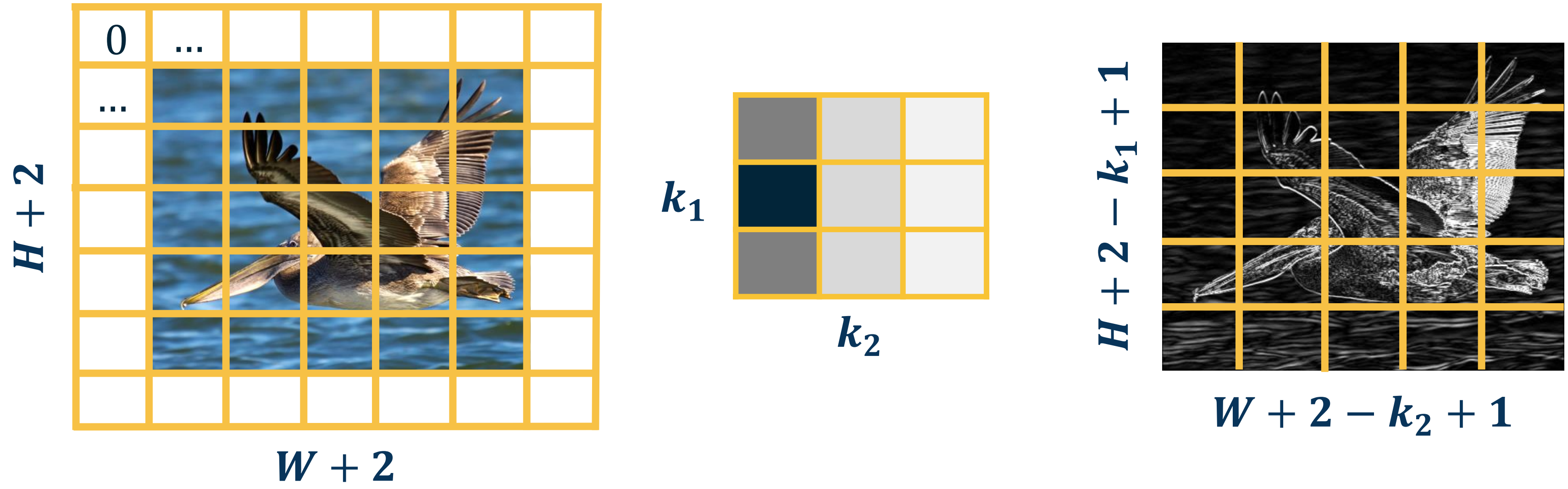
Convolutions are just **simple linear operations**

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** $(\mathbf{P = 1\ here})$

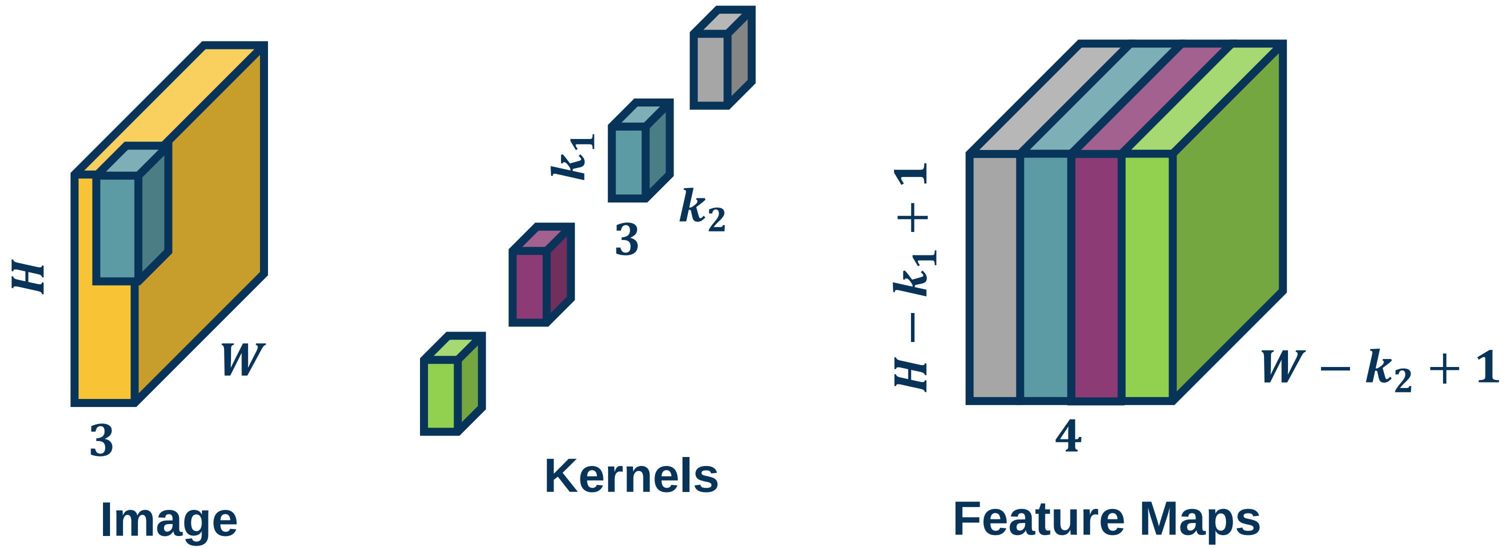


Adding Padding

We can have **multiple kernels per layer**

- We stack the feature maps together at the output

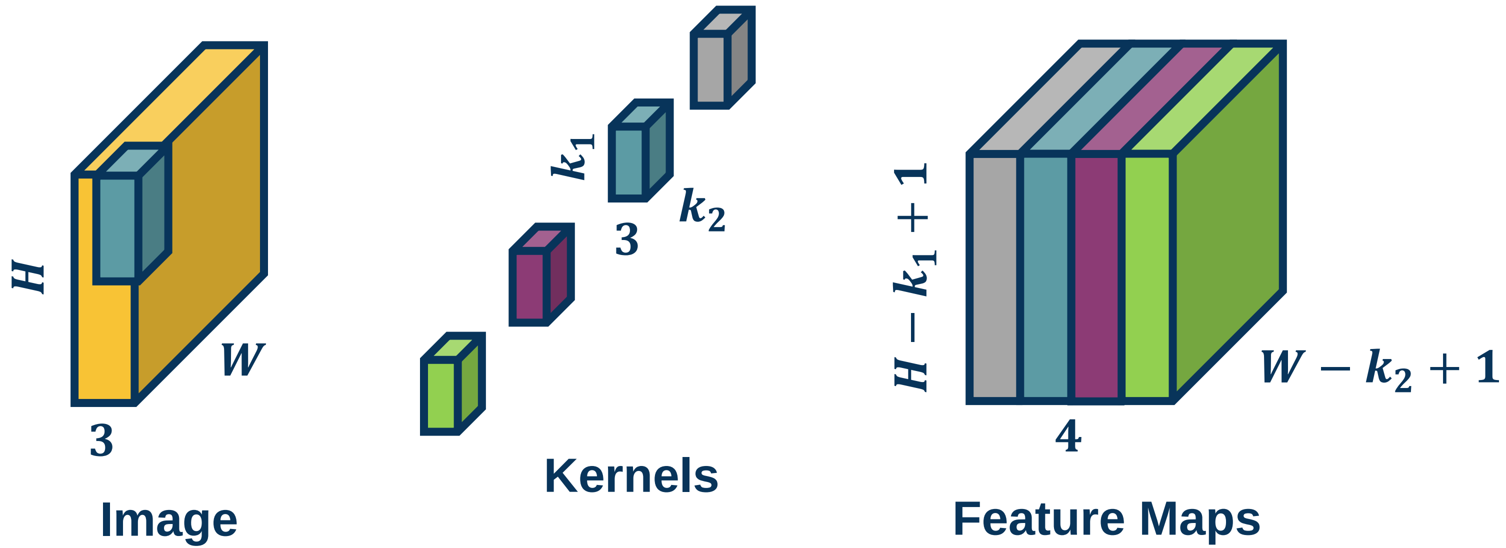
Number of channels in output is equal to *number of kernels*



Multiple Kernels

Number of parameters with N filters is: $\boldsymbol{N} * (\boldsymbol{k_1} * \boldsymbol{k_2} * \mathbf{3} + \mathbf{1})$

- Example:
  $\boldsymbol{k_1} = \mathbf{3}, \boldsymbol{k_2} = \mathbf{3},\ \boldsymbol{N} = \mathbf{4}\ \boldsymbol{input\ channels} = \mathbf{3}$, then $(\mathbf{3} * \mathbf{3} * \mathbf{3} + \mathbf{1}) * \mathbf{4}$ =112



Need to incorporate all upstream gradients:

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

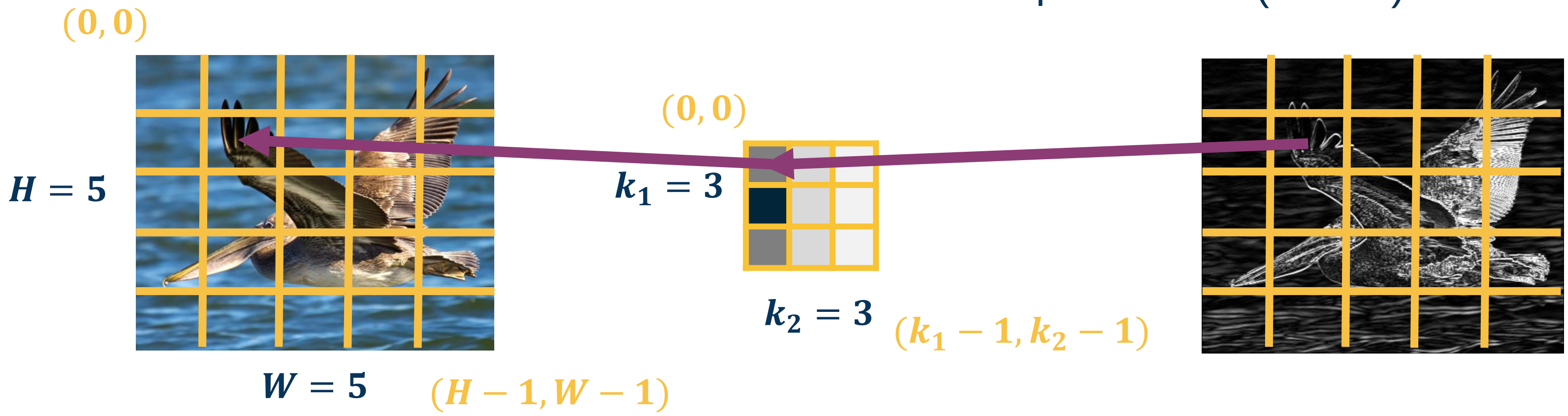
Chain Rule:

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

Sum over all output pixels

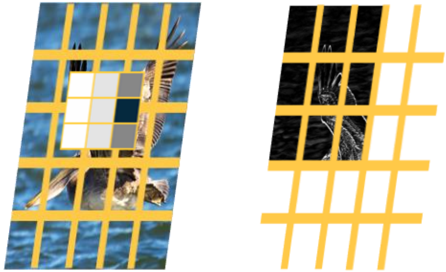
Upstream gradient (known)

We will compute



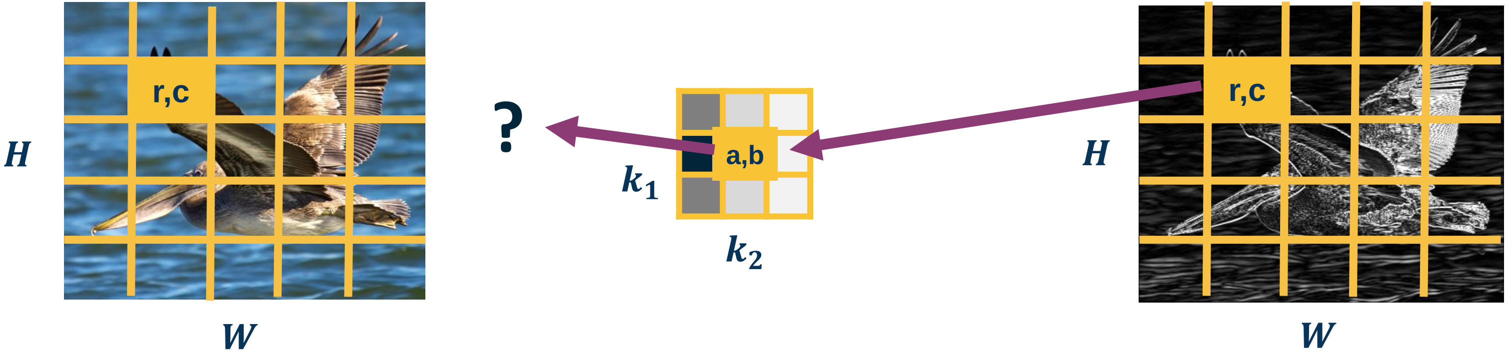
Chain Rule over all Output Pixels

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



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. $k(a,b)$
  - $k(0,0) * x(r,c)$, $k(1,1) * x(r+1,c+1)$, $k(2,2) * x(r+2,c+2)$ => 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

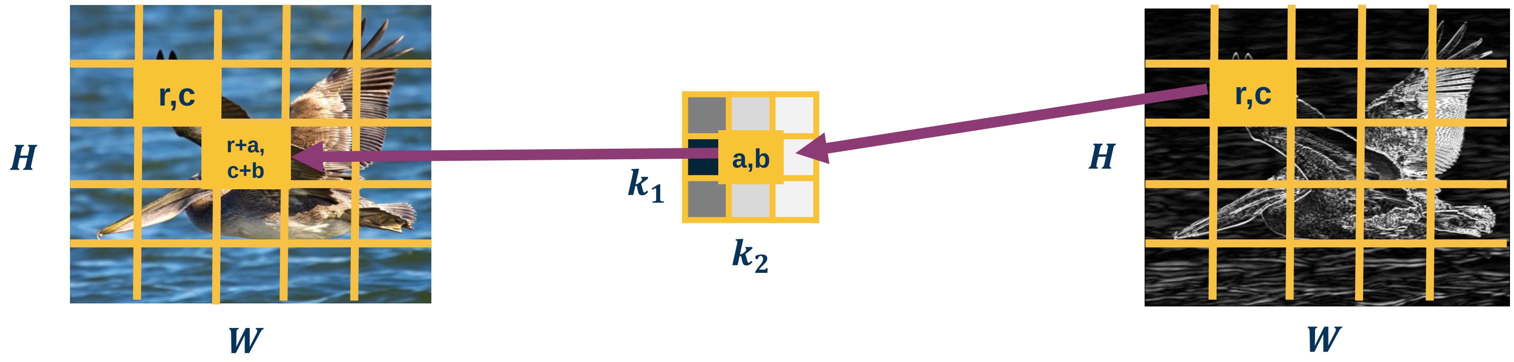
$$\frac{\partial y(r,c)}{\partial k(a,b)} = x(r+a, c+b)$$

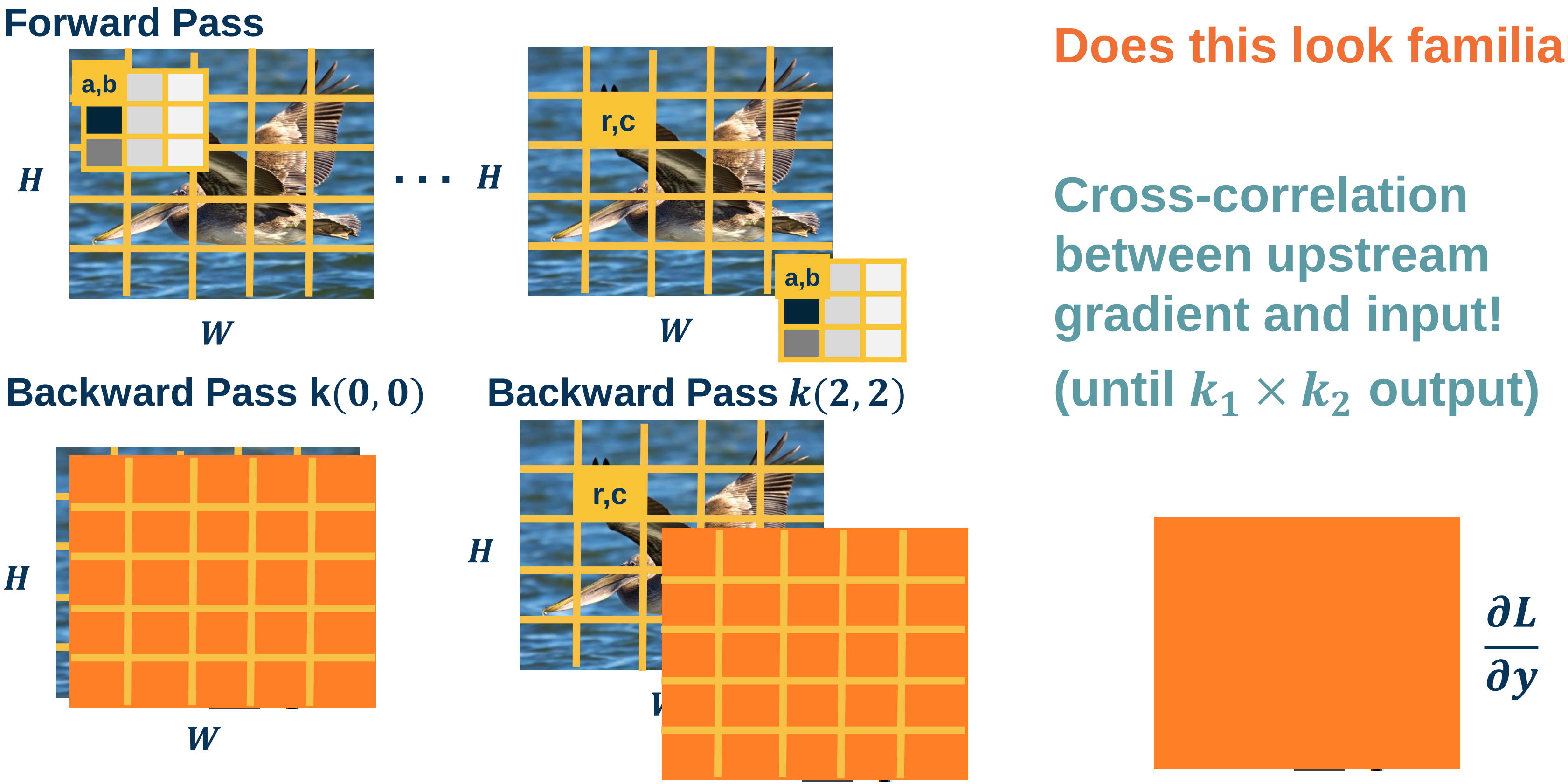
$$\frac{\partial L}{\partial k(a,b)} = \sum_{r=0}^{H-1} \sum_{c=0}^{W-1} \frac{\partial L}{\partial y(r,c)} x(r+a, c+b)$$

Does this look familiar?

Cross-correlation between upstream gradient and input!

(until $k_1 \times k_2$ output)





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)

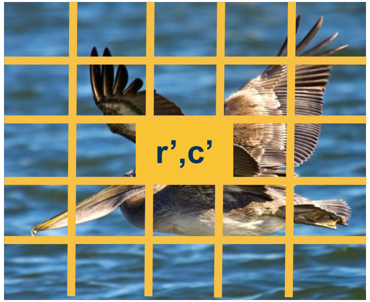
Calculate one pixel at a time $\frac{\partial L}{\partial x(r',c')}$

What does this input pixel affect at the output?

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

$(0,0)$

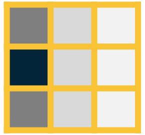
$H = 5$



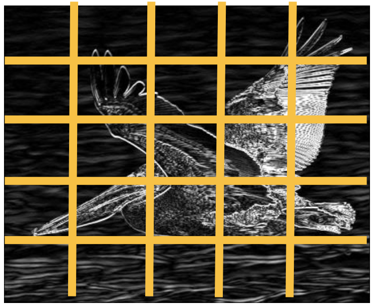
$W = 5$ $(H-1, W-1)$

$(0,0)$

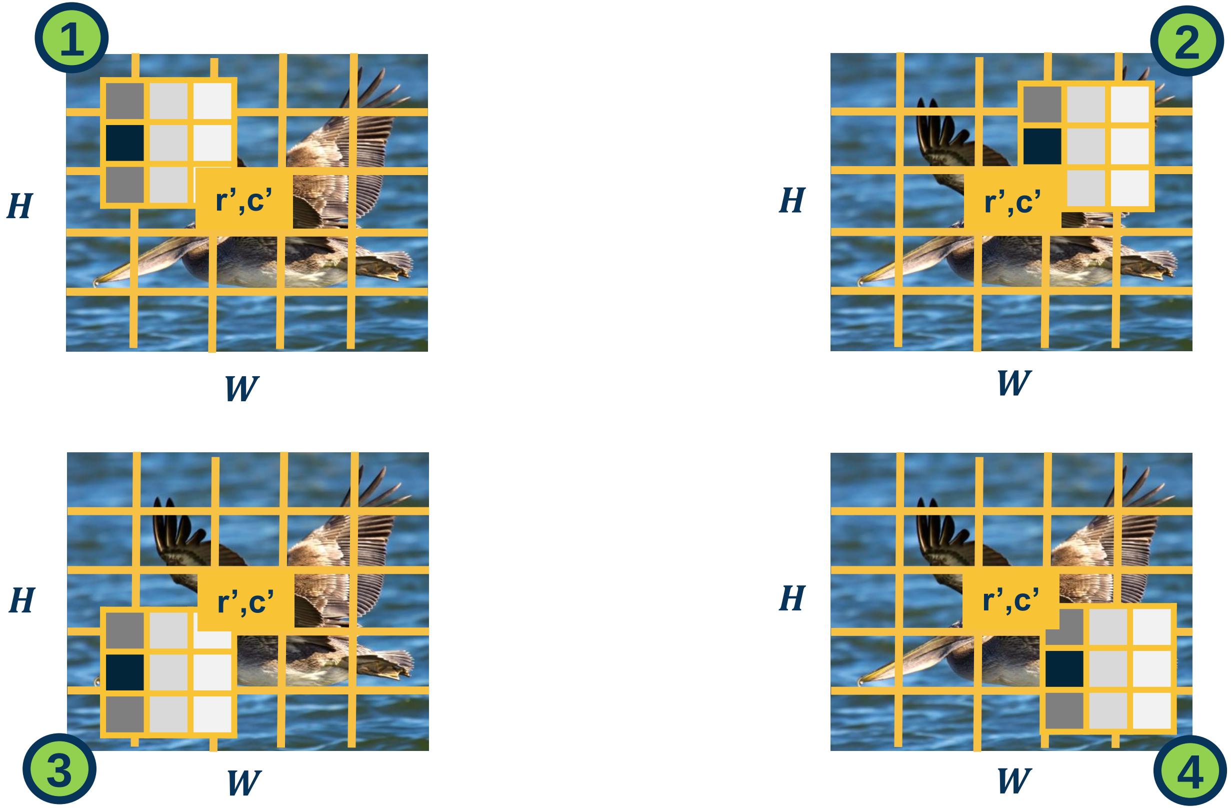
$k_1 = 3$



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

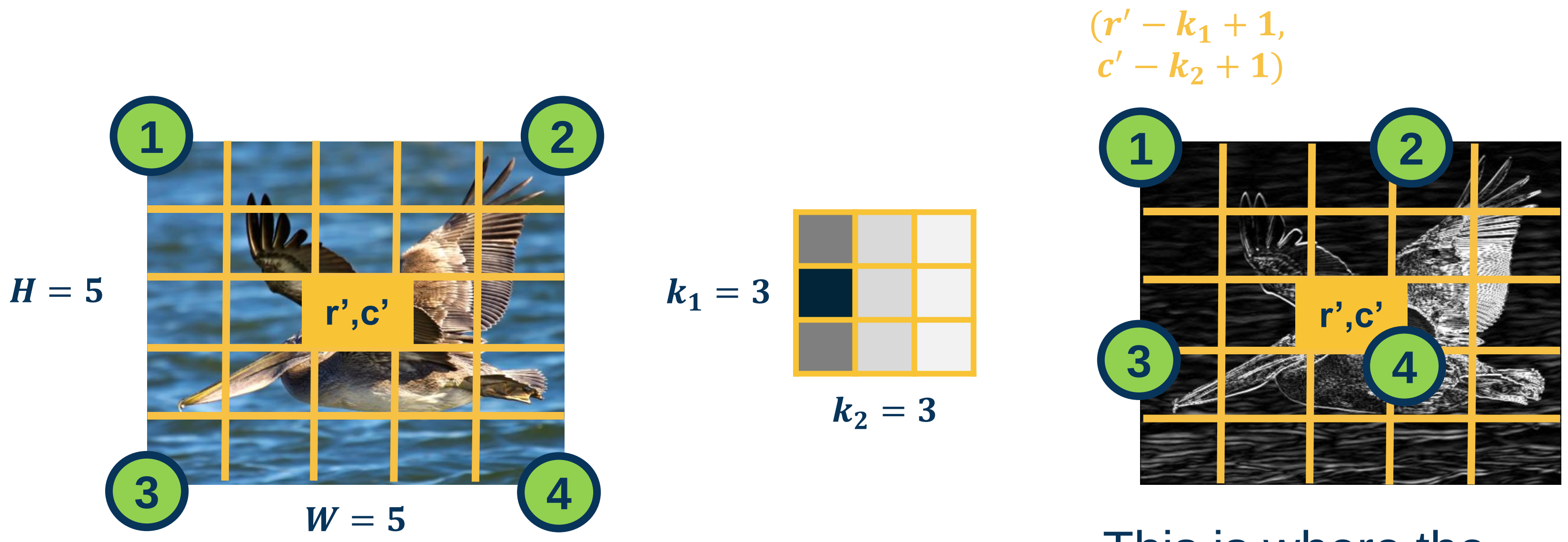


What an Input Pixel Affects at Output



Extents of Kernel Touching the Pixel

Extents at the Output



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

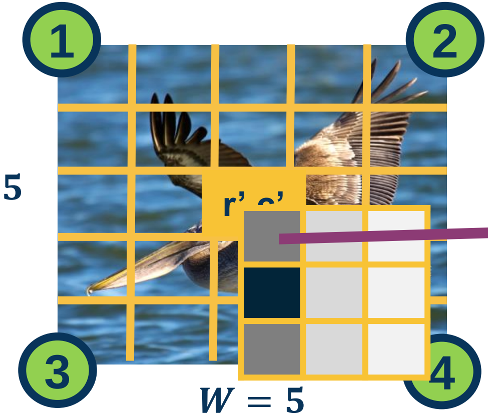
Chain rule for affected pixels (sum gradients):

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

$$x(r',c') * k(0,0) \Rightarrow y(r',c')$$
$$x(r',c') * k(1,1) \Rightarrow ?$$

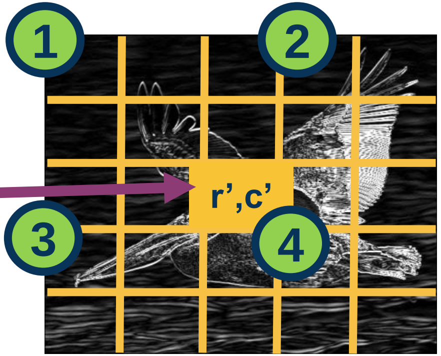
$$\frac{\partial L}{\partial x(r',c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(?,?)} \frac{\partial y(?,?)}{\partial x(r',c')}$$

$H = 5$



$W = 5$

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



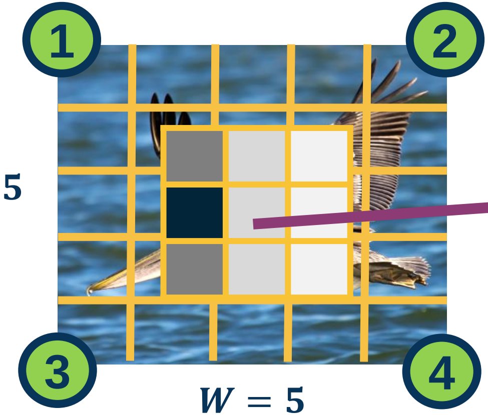
Chain rule for affected pixels (sum gradients):

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

$$x(r',c') * k(0,0) \Rightarrow y(r',c')$$
$$x(r',c') * k(1,1) \Rightarrow y(r'-1,c'-1)$$
$$\dots$$
$$x(r',c') * k(a,b) \Rightarrow y(r'-a,c'-b)$$

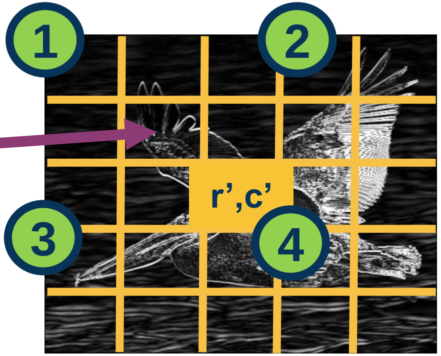
$$\frac{\partial L}{\partial x(r',c')} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} \frac{\partial L}{\partial y(?,?)} \frac{\partial y(?,?)}{\partial x(r',c')}$$

$H = 5$



$W = 5$

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

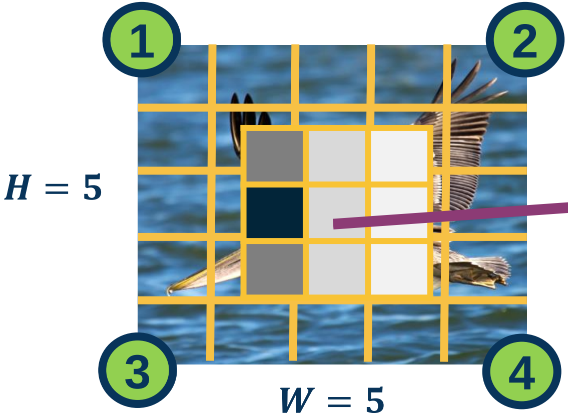


Chain rule for affected pixels (sum gradients):

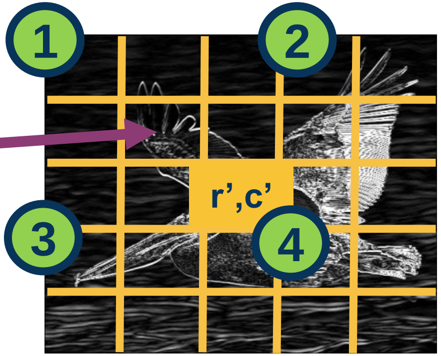
$$\frac{\partial L}{\partial x(r',c')} = \sum_{Pixels\ p} \frac{\partial L}{\partial y(p)} \frac{\partial y(p)}{\partial x(r',c')}$$

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

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



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



Summing Gradient Contributions

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

$$\boldsymbol{y}(\boldsymbol{r}',\boldsymbol{c}') = (\boldsymbol{x} * \boldsymbol{k})(\boldsymbol{r}',\boldsymbol{c}') = \sum_{\boldsymbol{a}'=0}^{\boldsymbol{k}_1-1} \sum_{\boldsymbol{b}'=0}^{\boldsymbol{k}_2-1} \boldsymbol{x}(\boldsymbol{r}'+\boldsymbol{a}',\boldsymbol{c}'+\boldsymbol{b}')\,\boldsymbol{k}(\boldsymbol{a}',\boldsymbol{b}')$$

Plug in what we actually wanted :

$$\boldsymbol{y}(\boldsymbol{r}'-\boldsymbol{a},\boldsymbol{c}'-\boldsymbol{b}) = (\boldsymbol{x} * \boldsymbol{k})(\boldsymbol{r}',\boldsymbol{c}') = \sum_{\boldsymbol{a}'=0}^{\boldsymbol{k}_1-1} \sum_{\boldsymbol{b}'=0}^{\boldsymbol{k}_2-1} \boldsymbol{x}(\boldsymbol{r}'-\boldsymbol{a}+\boldsymbol{a}',\boldsymbol{c}'-\boldsymbol{b}+\boldsymbol{b}')\,\boldsymbol{k}(\boldsymbol{a}',\boldsymbol{b}')$$

What is $\dfrac{\partial \boldsymbol{y}(\boldsymbol{r}'-\boldsymbol{a},\boldsymbol{c}'-\boldsymbol{b})}{\partial \boldsymbol{x}(\boldsymbol{r}',\boldsymbol{c}')} = \mathbf{k}(\boldsymbol{a},\boldsymbol{b})$

(we want term with $\boldsymbol{x}(\boldsymbol{r}',\boldsymbol{c}')$ in it;
this happens when $\mathbf{a} = \mathbf{a}'$ and $\mathbf{b} = \mathbf{b}'$)

Plugging in to earlier equation:

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

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

Does this look familiar?

Convolution between upstream gradient and kernel!

(can implement by flipping kernel and cross- correlation)

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

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)

Summary

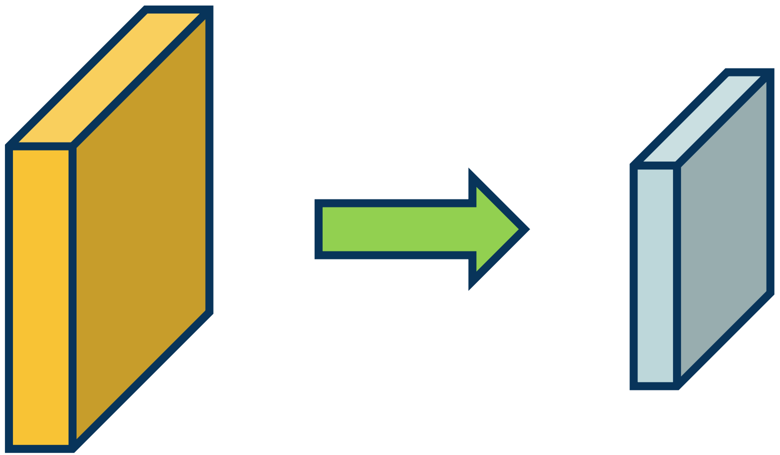
Georgia Tech

Pooling Layers

Georgia Tech

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

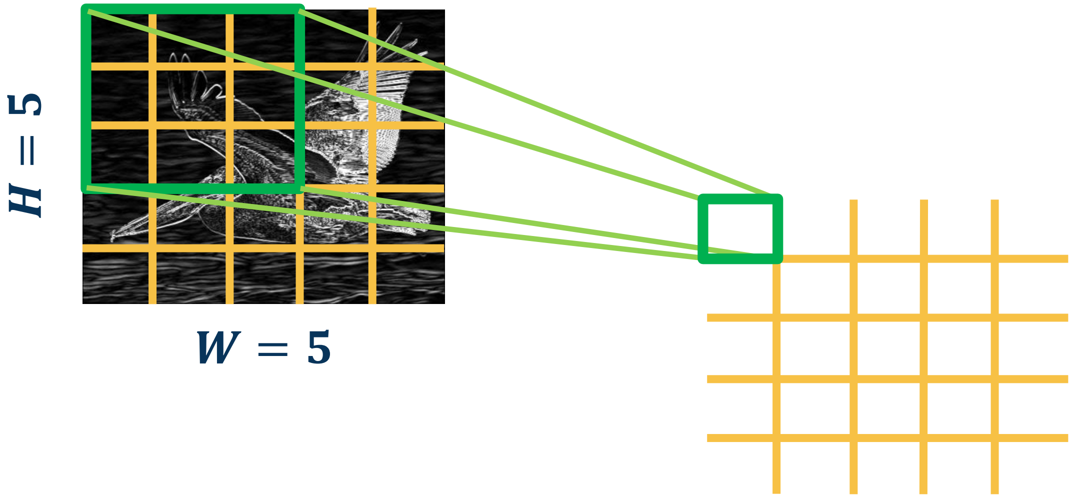
- **kernel_size** – the size of the window to take a max over
- **stride** – the stride of the window. Default value is `kernel_size`
- **padding** – implicit zero padding to be added on both sides

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**

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



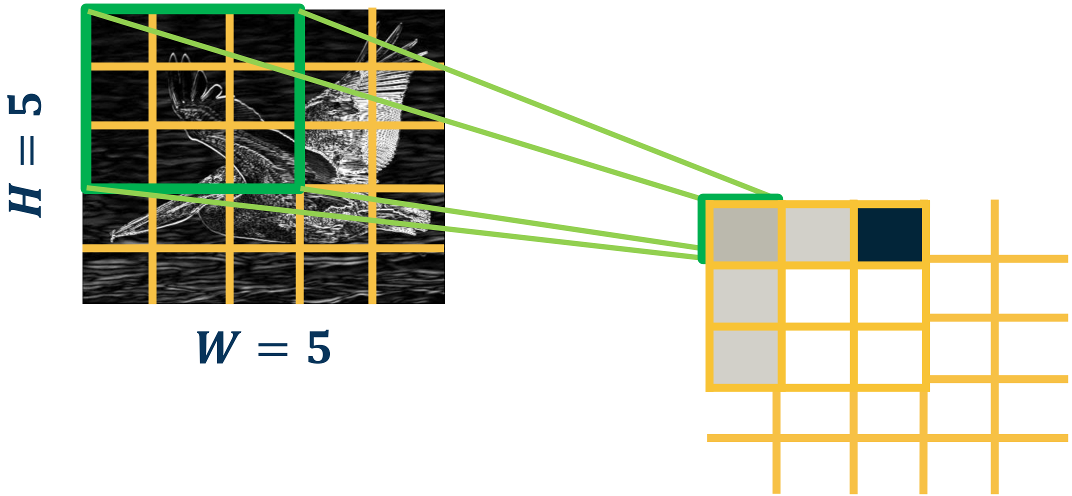
How many learned parameters does this layer have?

None!

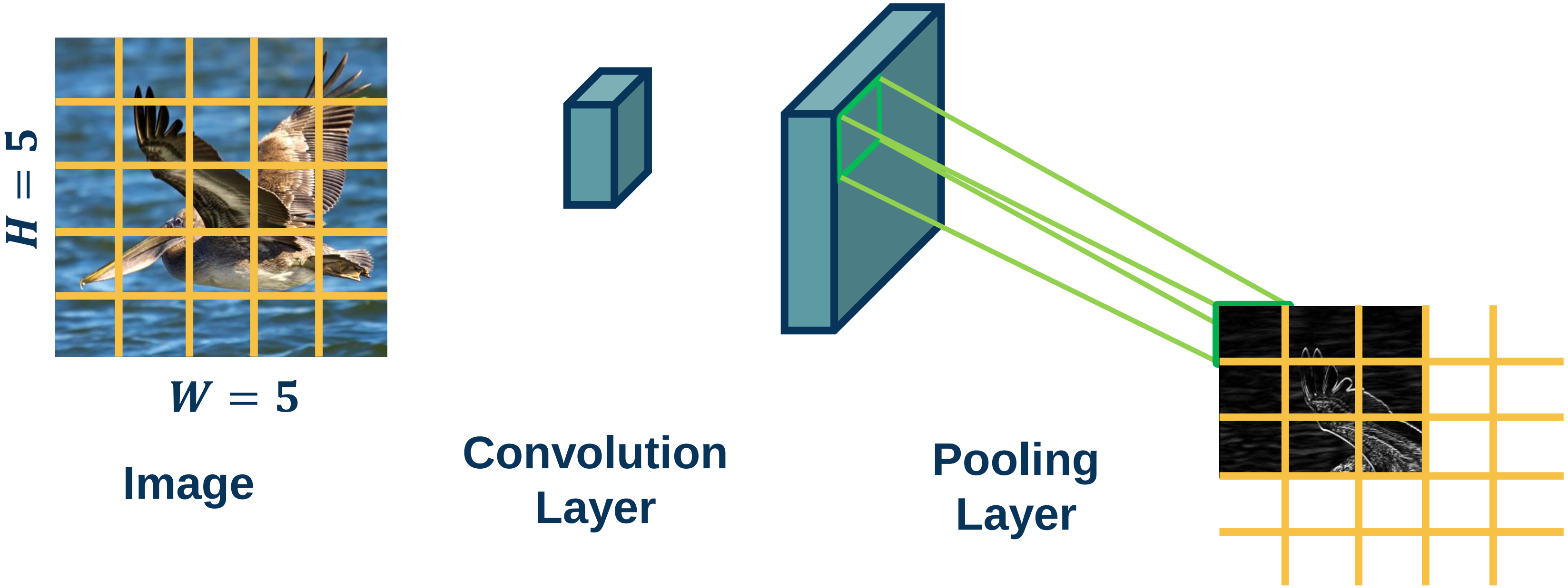
Not restricted to max; can use any differentiable function

- Not very common in practice

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

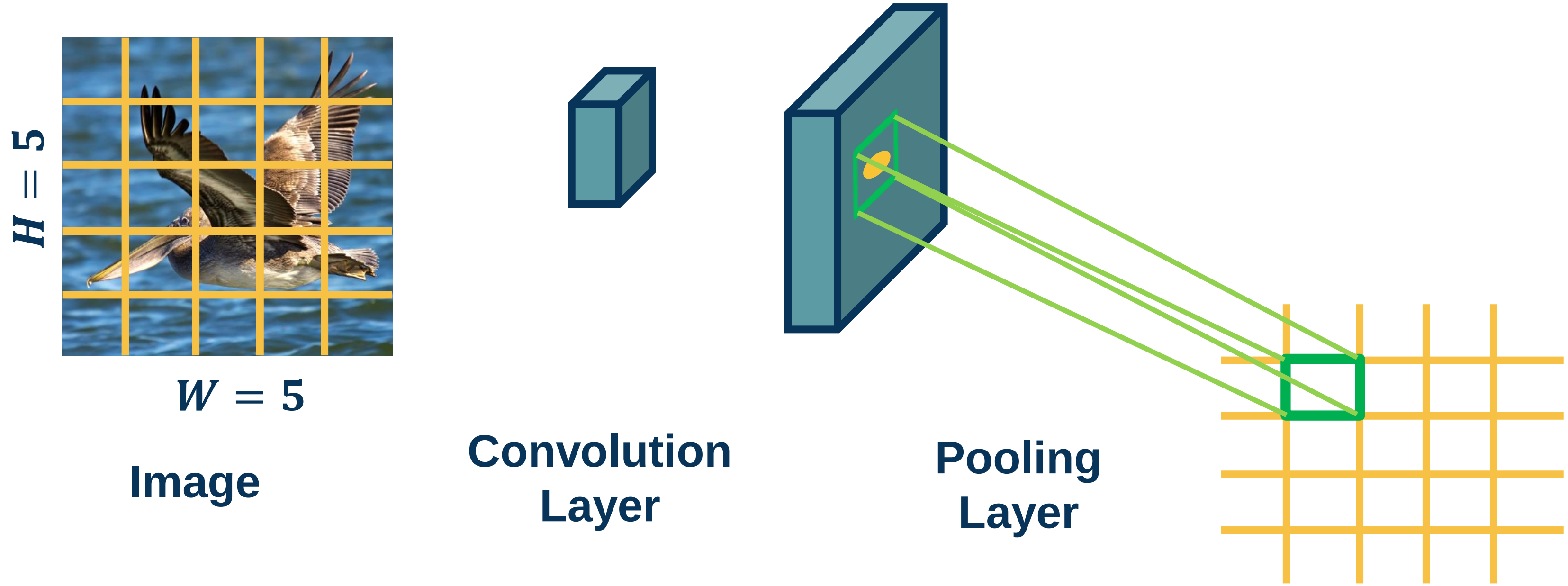


Since the **output** of convolution and pooling layers are **(multi-channel) images**, we can sequence them just as any other layer



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

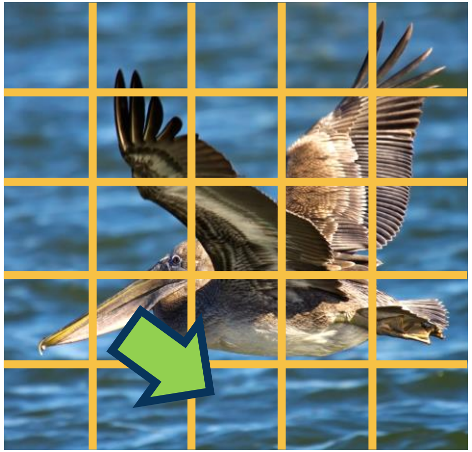
- If feature (such as beak) translated a little bit, output values still **remain the same**



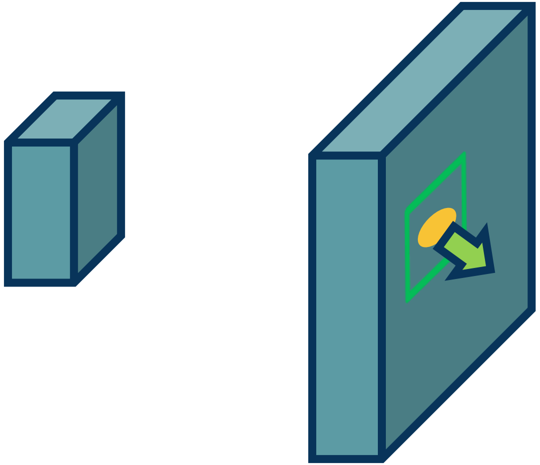
Convolution by itself has the property of **equivariance**

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

$H = 5$



$W = 5$

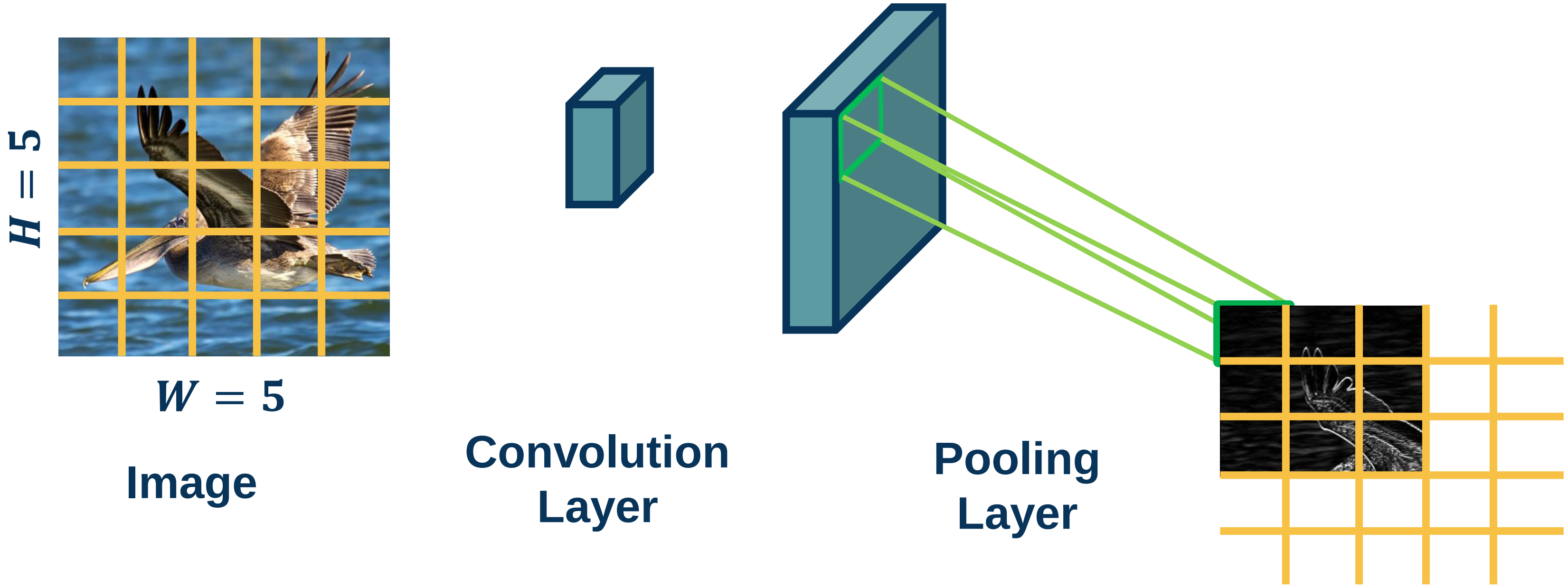


Invariance vs. Equivariance

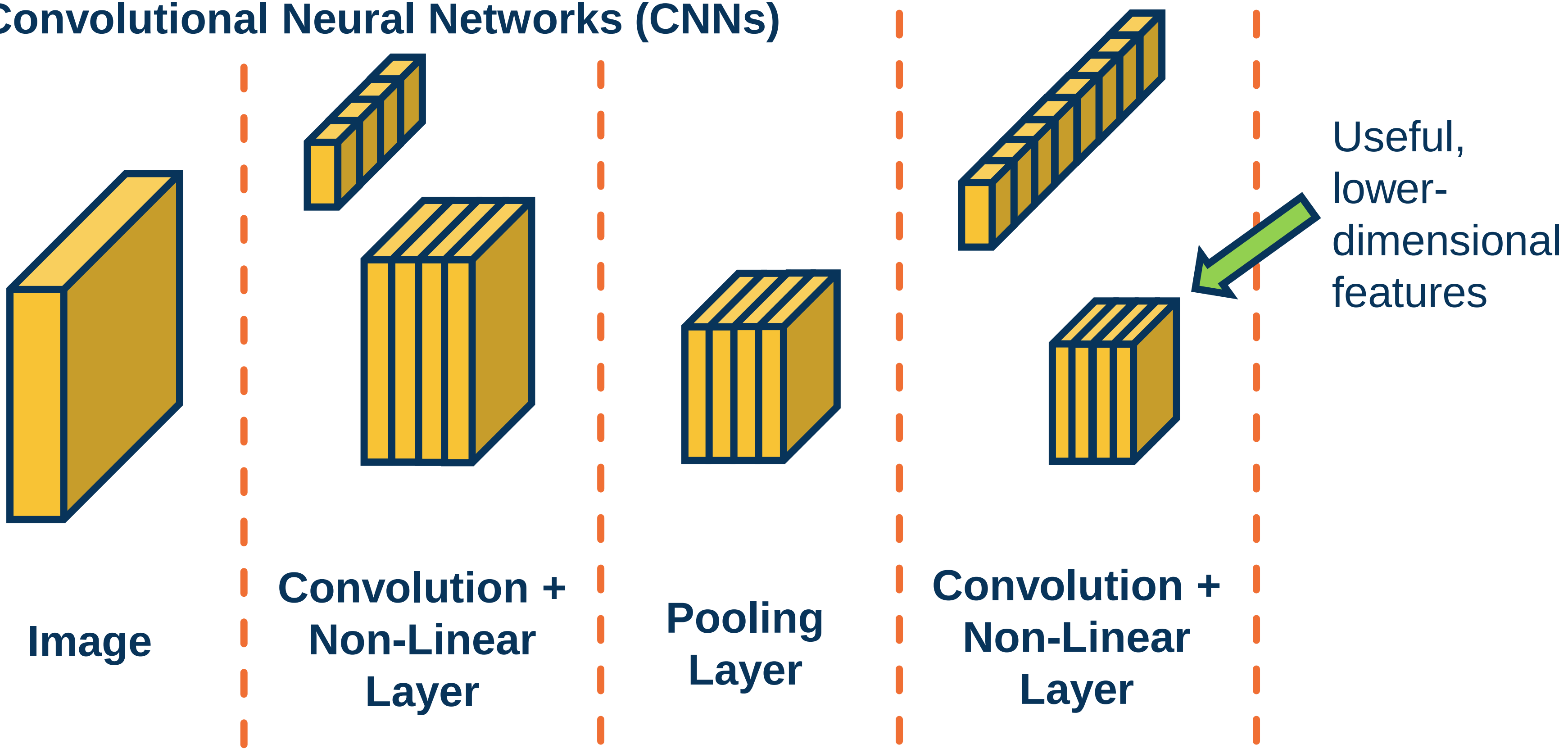
Simple Convolutional Neural Networks

Georgia Tech

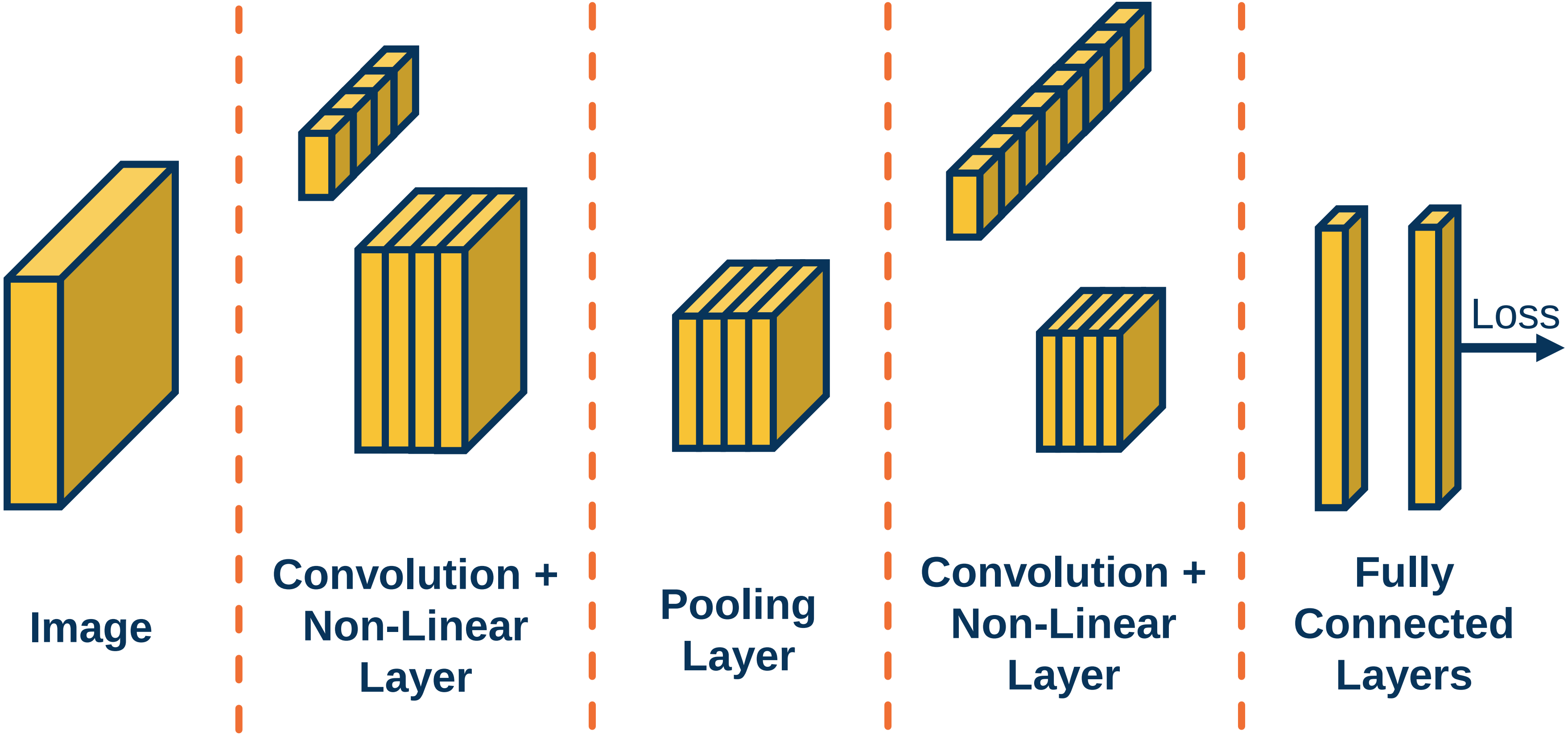
Since the **output** of convolution and pooling layers are **(multi-channel) images**, we can sequence them just as any other layer



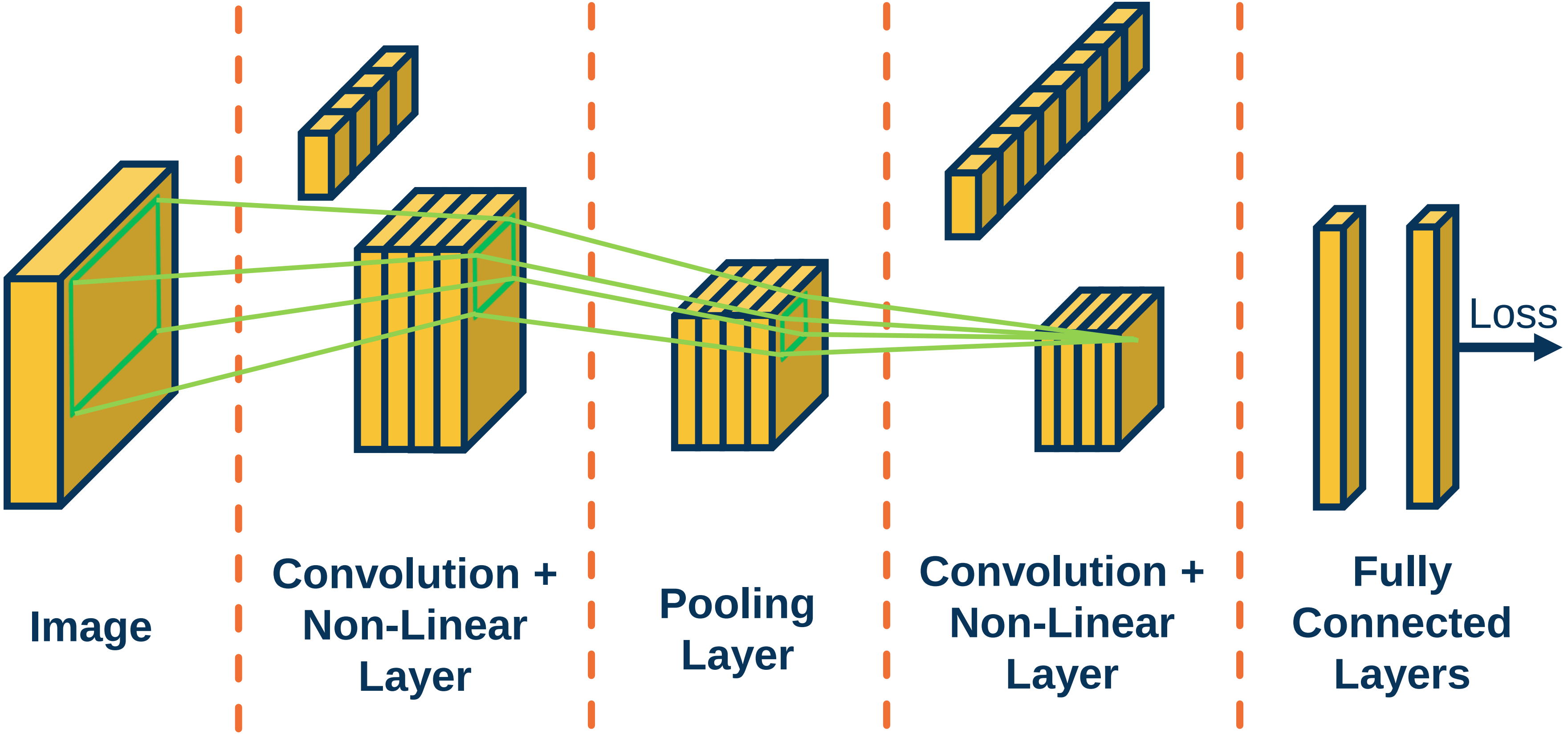
Convolutional Neural Networks (CNNs)



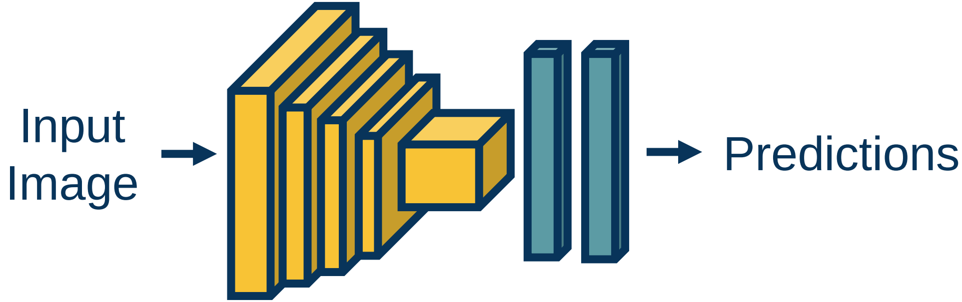
Alternating Convolution and Pooling



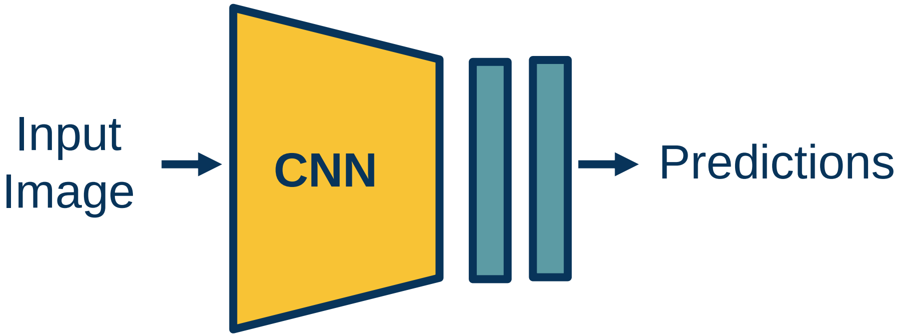
Adding a Fully Connected Layer



Receptive Fields



Convolutional Neural Networks



Typical Depiction of CNNs

These architectures have existed **since 1980s**

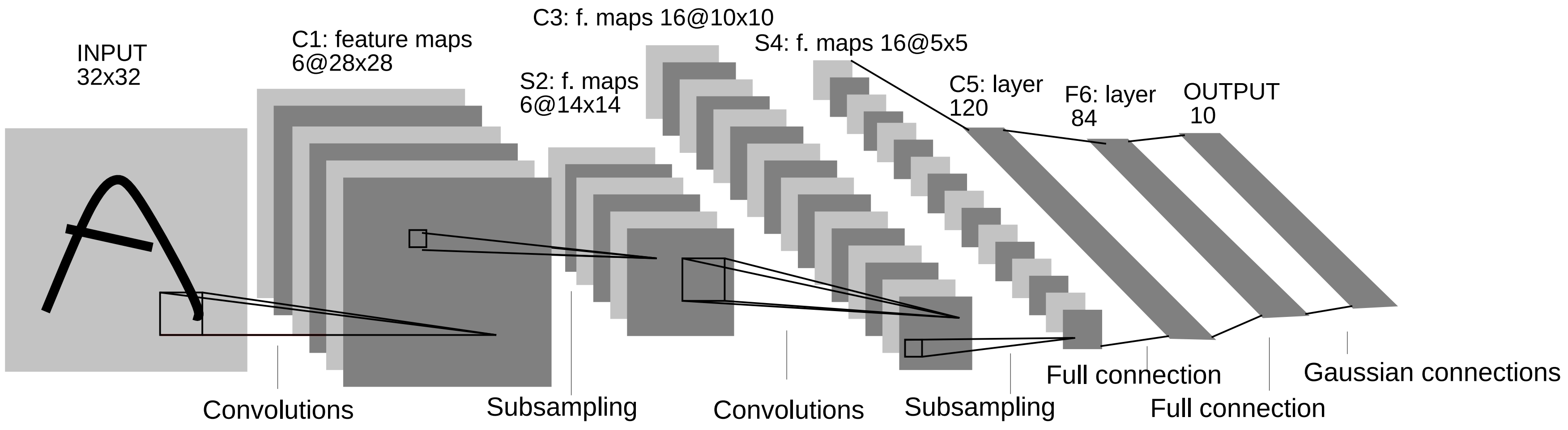


Image Credit: Yann LeCun, Kevin Murphy

Handwriting Recognition

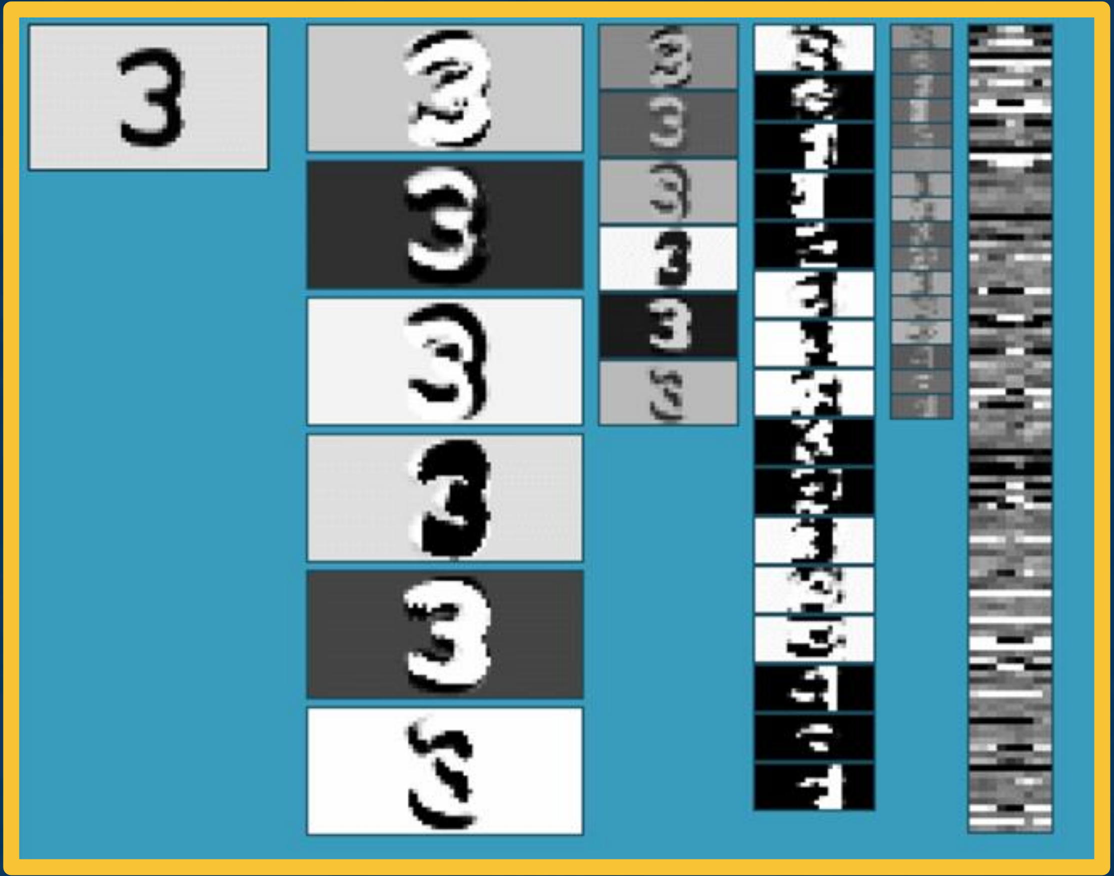


Image Credit: Yann LeCun

Translation Equivariance (Conv Layers) & Invariance (Output)

Image Credit: Yann LeCun

(Some) Rotation Invariance

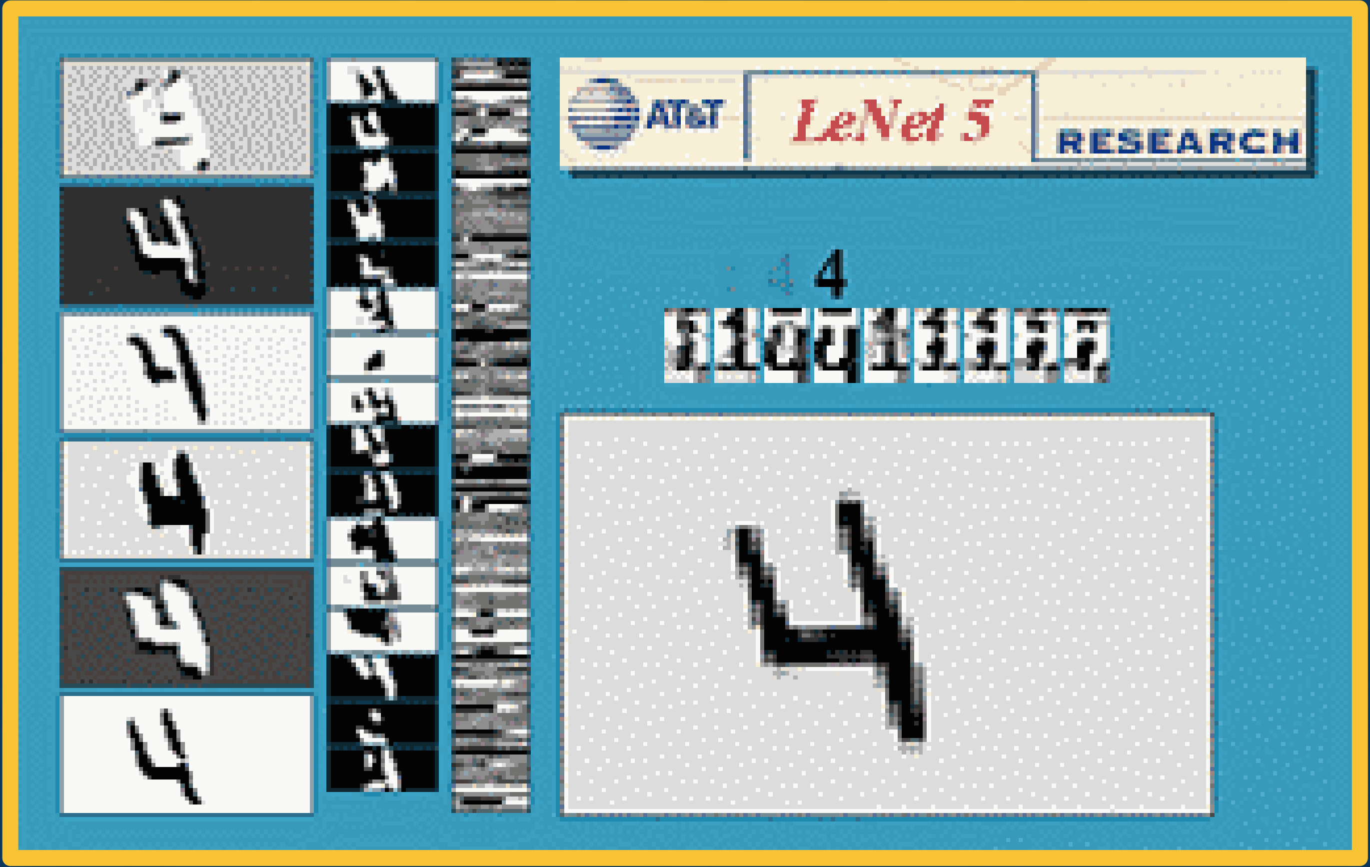


Image Credit: Yann LeCun

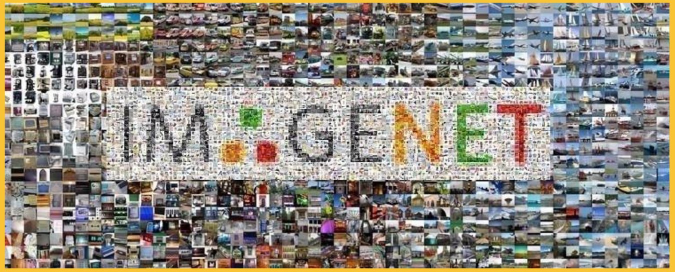
(Some) Scale Invariance

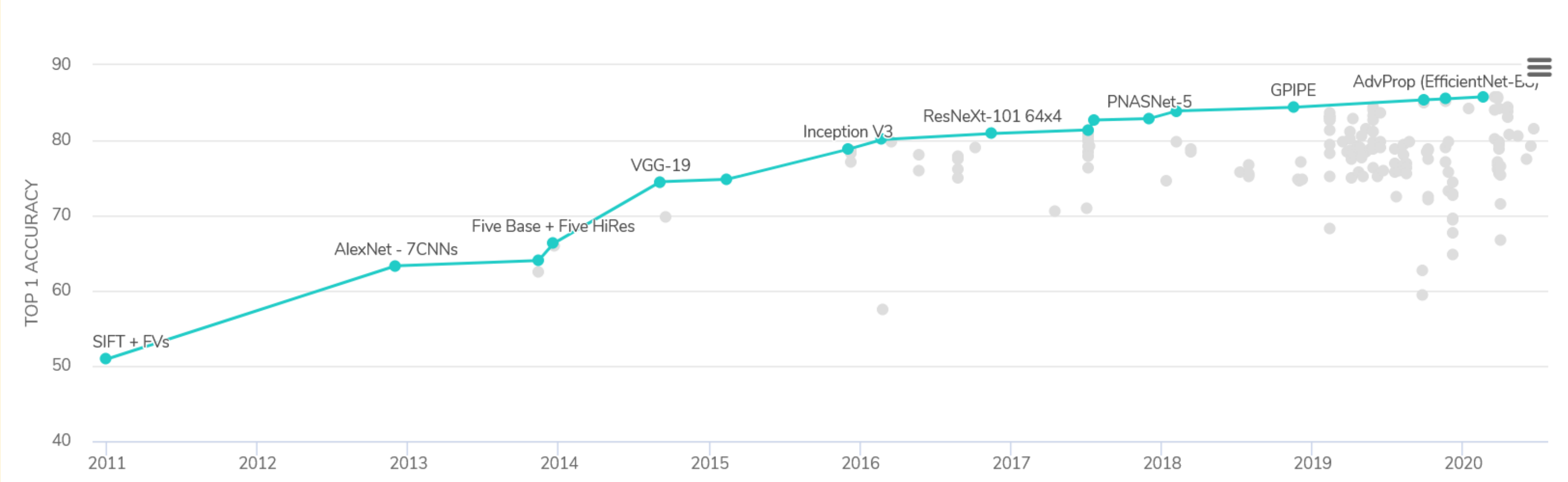
Image Credit: Yann LeCun

Advanced Convolutional Networks

Georgia Tech

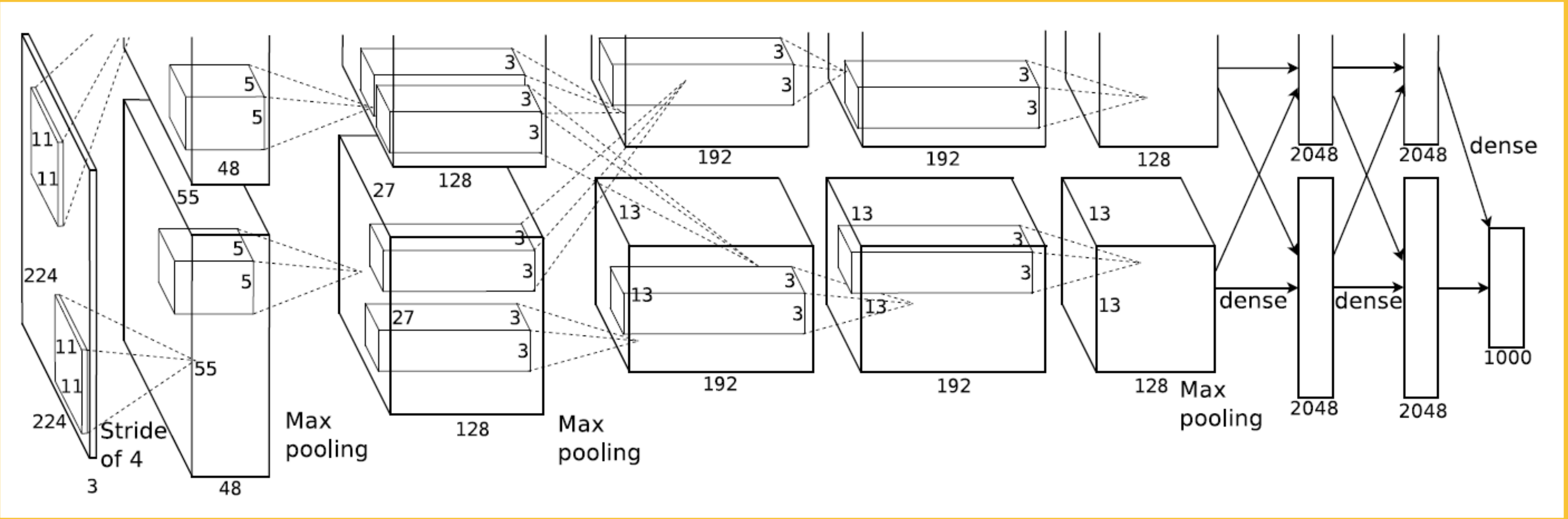
The Importance of Benchmarks





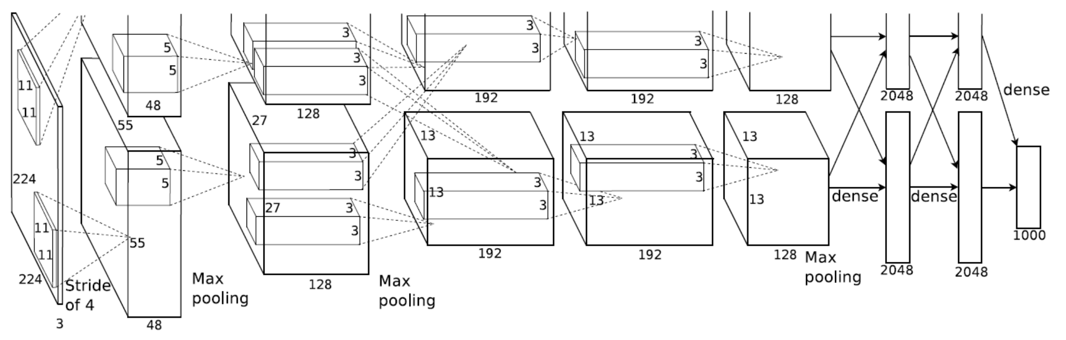
From: https://paperswithcode.com

AlexNet - Architecture



From: Krizhevsky et al., ImageNet Classification with Deep ConvolutionalNeural Networks, 2012.

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



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

Georgia Tech

(not counting biases)

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

ConvNet Configuration					
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**
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

VGG

(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0

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

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

POOL2: [112x112x64] memory: 112*112*64=800K params: 0

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

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

POOL2: [56x56x128] memory: 56*56*128=400K params: 0

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

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

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

POOL2: [28x28x256] memory: 28*28*256=200K params: 0

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

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

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

POOL2: [14x14x512] memory: 14*14*512=100K params: 0

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

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

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

POOL2: [7x7x512] memory: 7*7*512=25K params: 0

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

FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216

FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

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

VGG – Key Characteristics

Key aspects:

Repeated application of:

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

Very large number of parameters

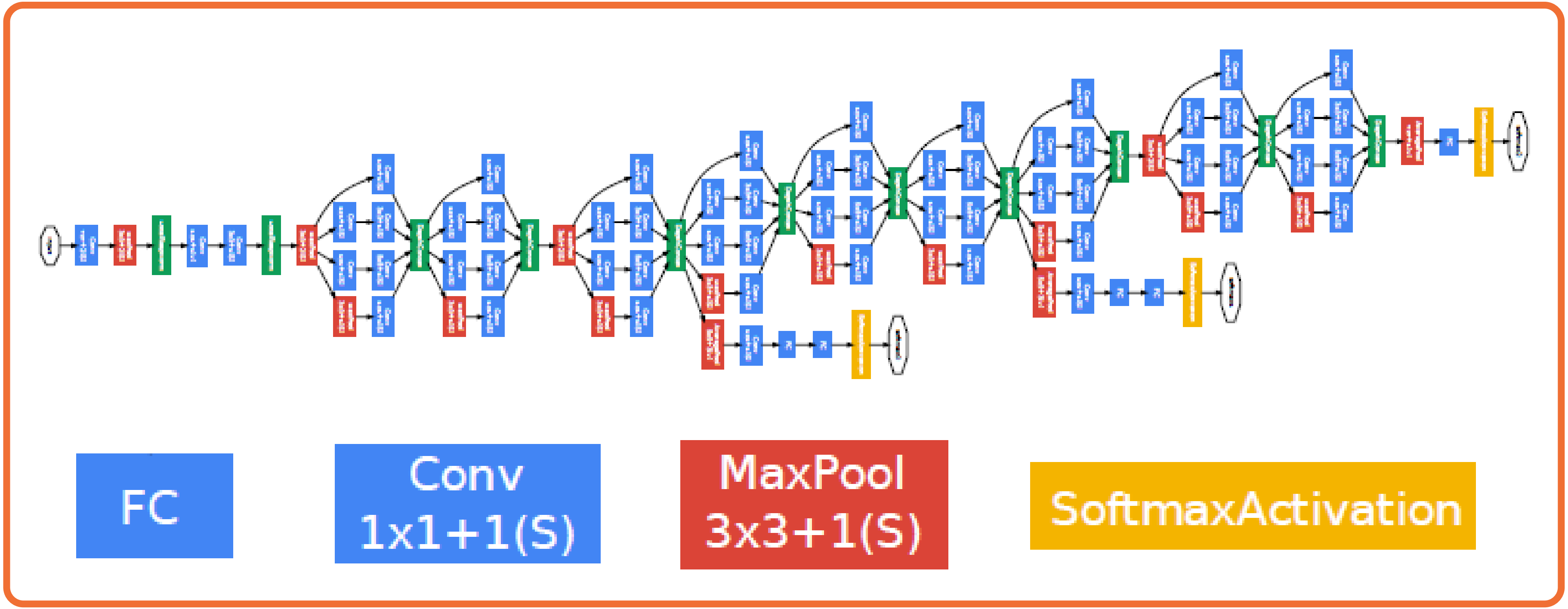
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128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

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

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

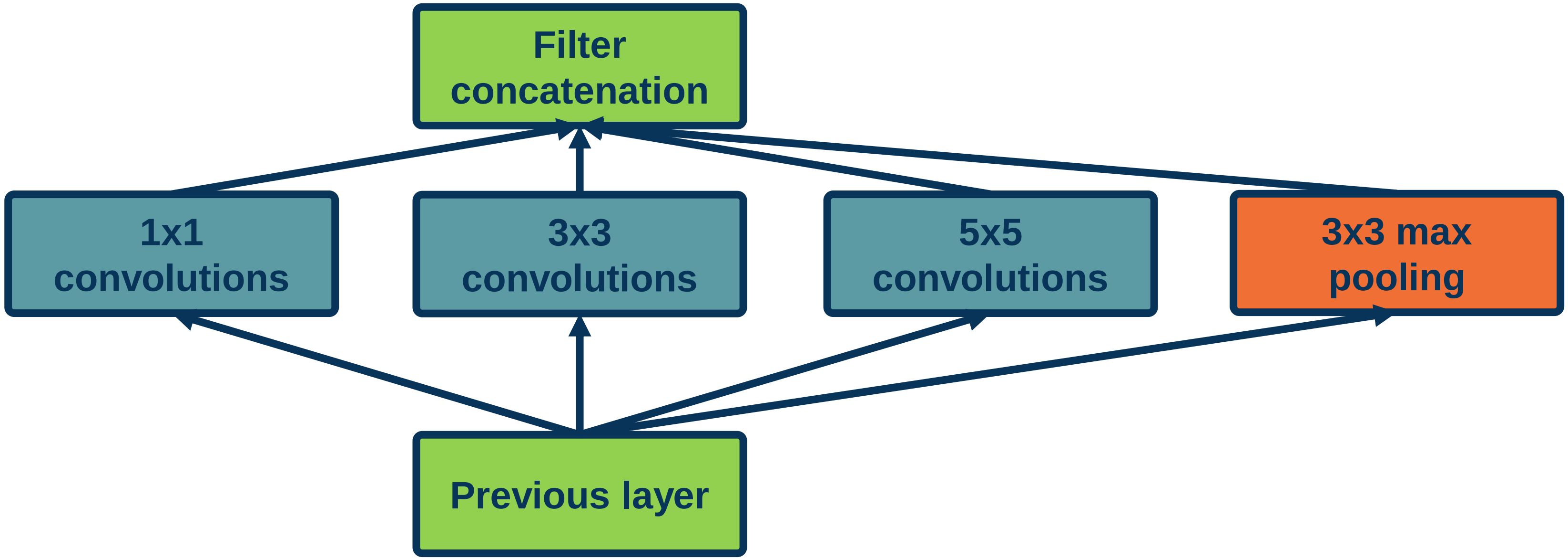
From: Simonyan & Zimmerman, Very Deep Convolutional Networks for Large-Scale Image Recognition
From: Slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

But have become **deeper and more complex**



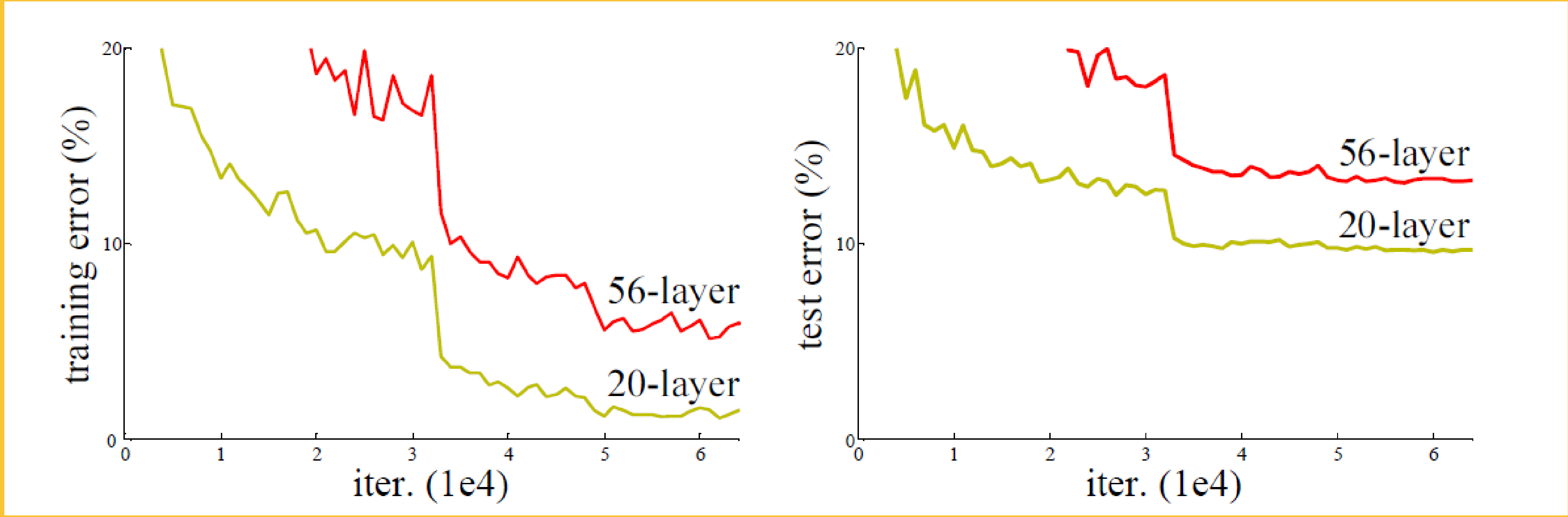
From: Szegedy et al. Going deeper with convolutions

Key idea: Repeated blocks and multi-scale features



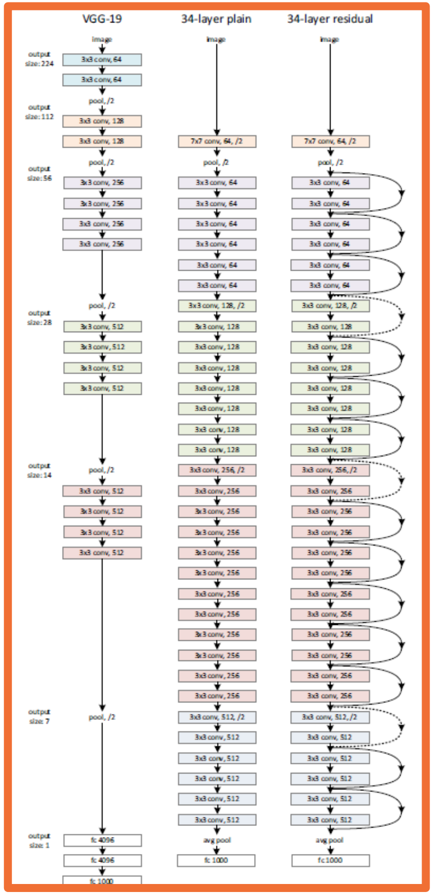
From: Szegedy et al. Going deeper with convolutions

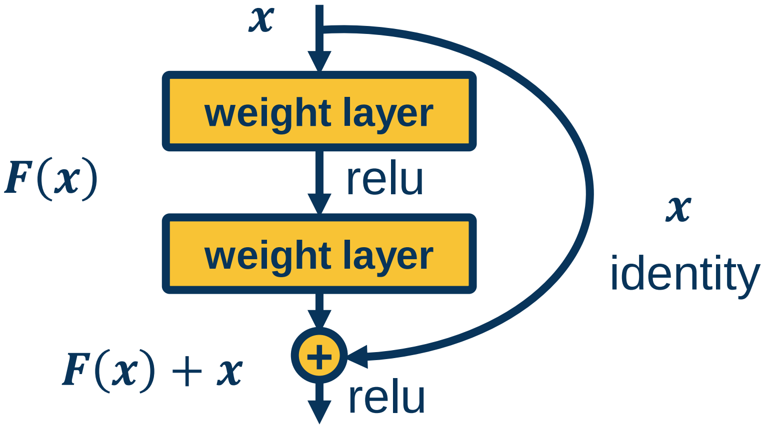
The Challenge of Depth



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)

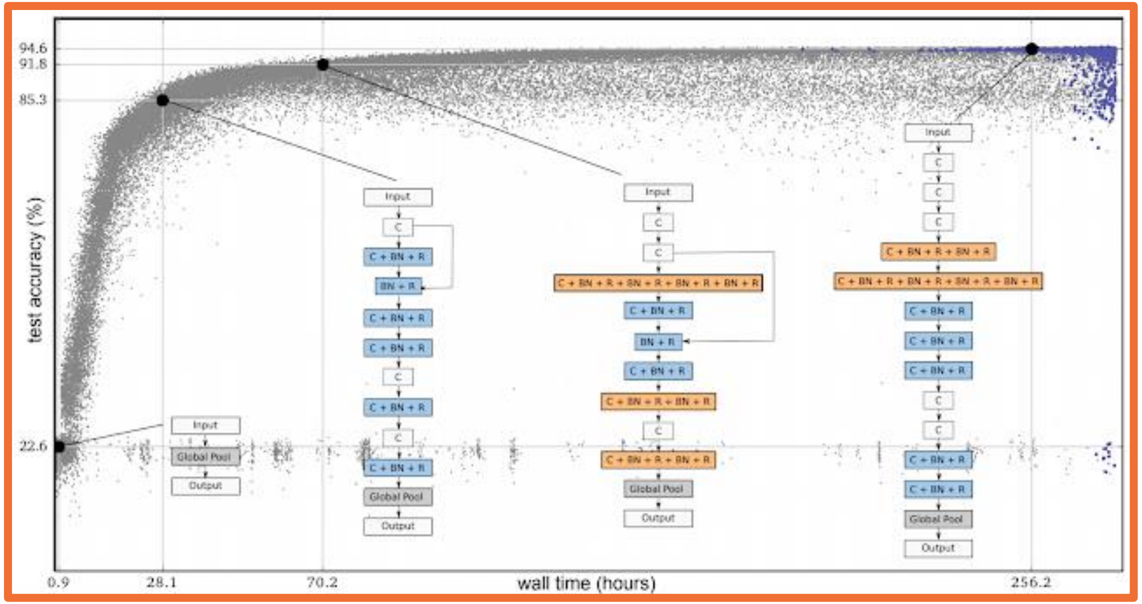
Same is true for gradients!

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

Residual Blocks and Skip Connections

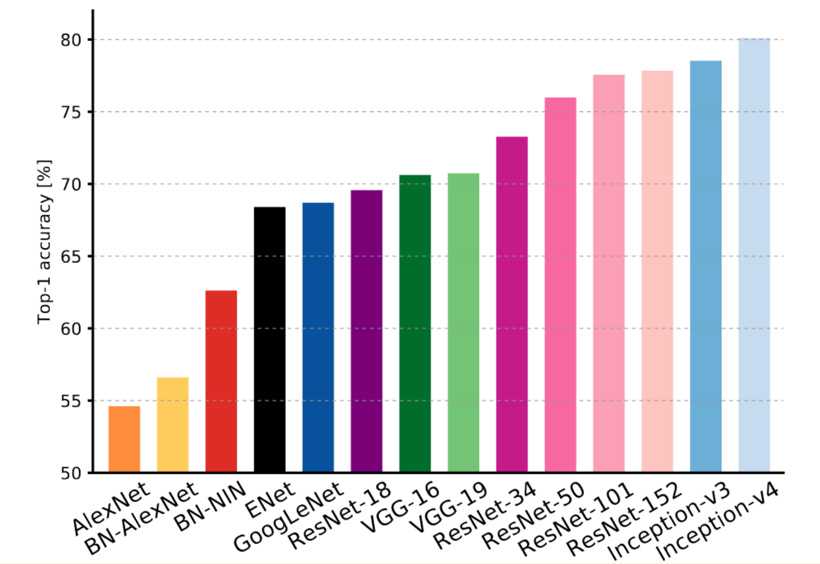
Several ways to *learn* architectures:

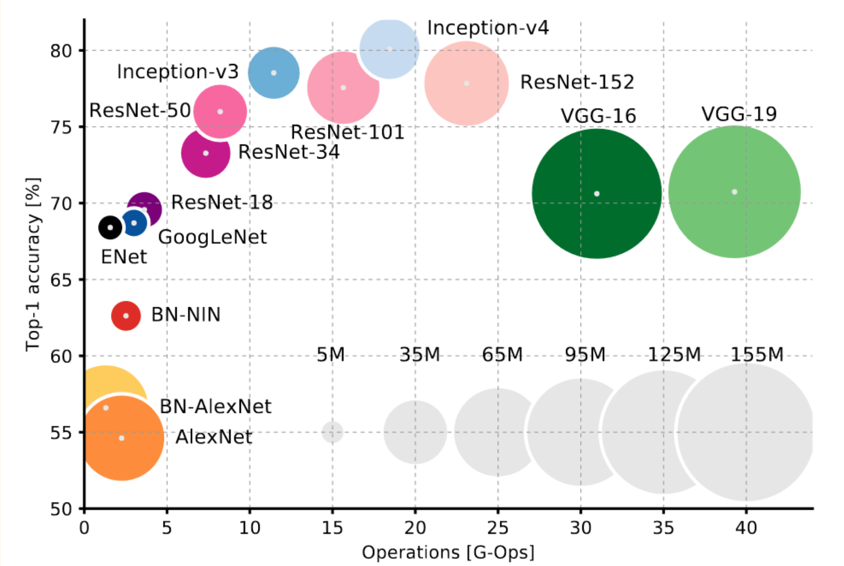
- Evolutionary learning and reinforcement learning
- Prune over-parameterized networks
- Learning of **repeated blocks** typical



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

Georgia Tech