Topics:

- Convolutional Neural Networks

CS 4803-DL / 7643-A

ZSOLT KIRA

- **Assignment 2**
  - Implement convolutional neural networks

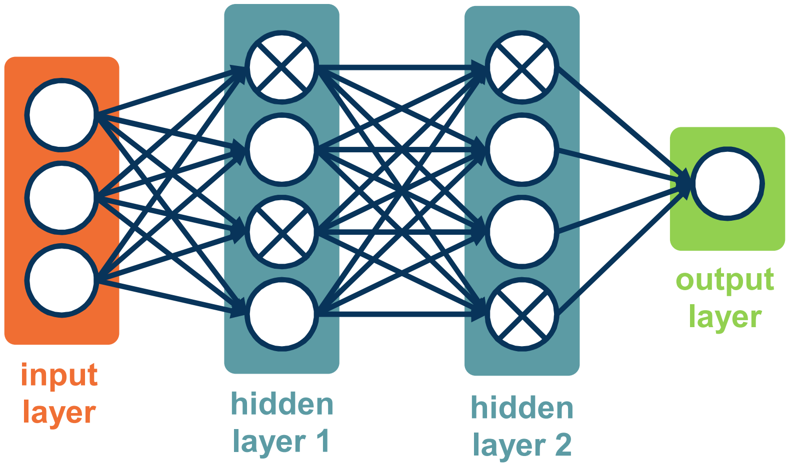
- **GPU resources**
  - **Google Cloud Credits**
  - **Google Colab**

Interpretation 1: The model should not rely too heavily on particular features

- If it does, it has probability $1 - p$ of losing that feature in an iteration

Interpretation 2: Training 2^n networks:

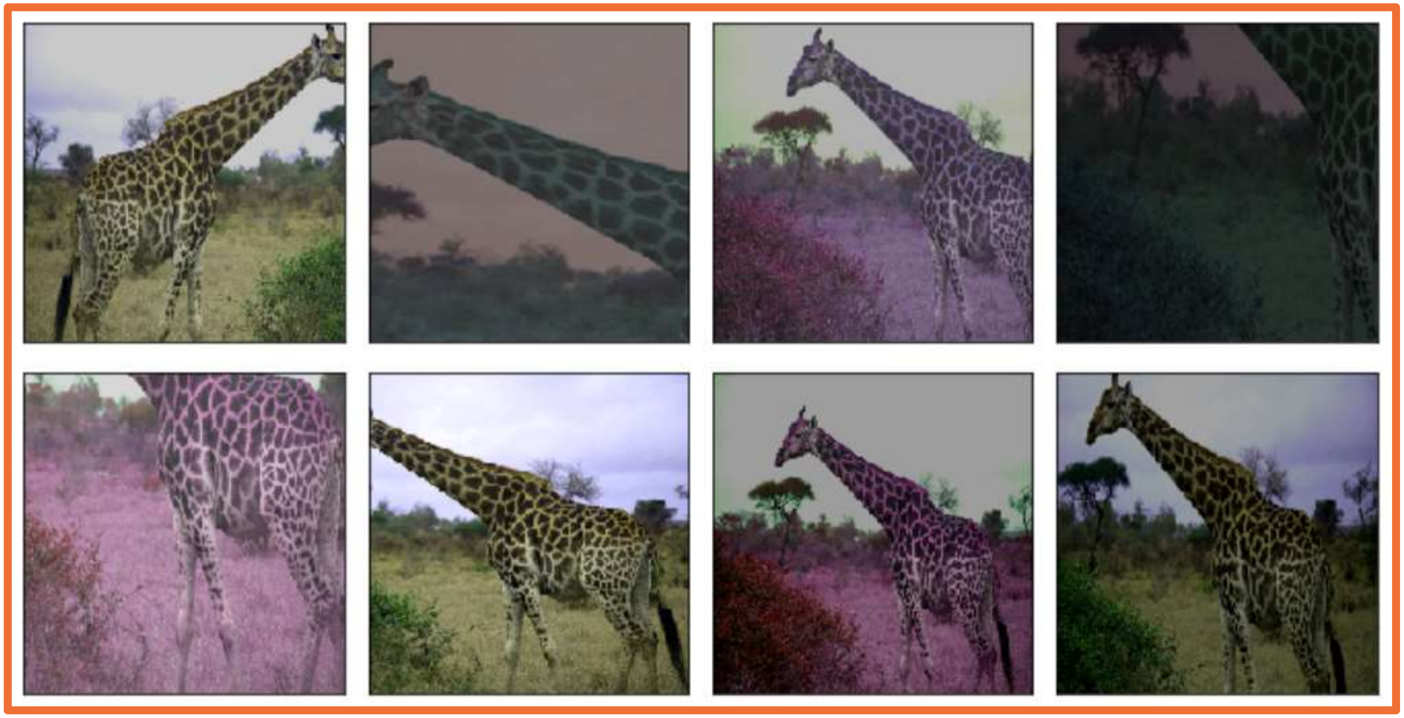
- Each configuration is a network
- Most are trained with 1 or 2 mini-batches of data



From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Why Dropout Works

We can **combine these transformations** to add even more variety!



From https://mxnet.apache.org/versions/1.5.0/tutorials/gluon/data_augmentation.html

Combining Transformations

Class Imbalance: Focal Loss

Cross Entropy: easy examples incur a non-negligible loss, which in aggregate mask out the harder, rare examples

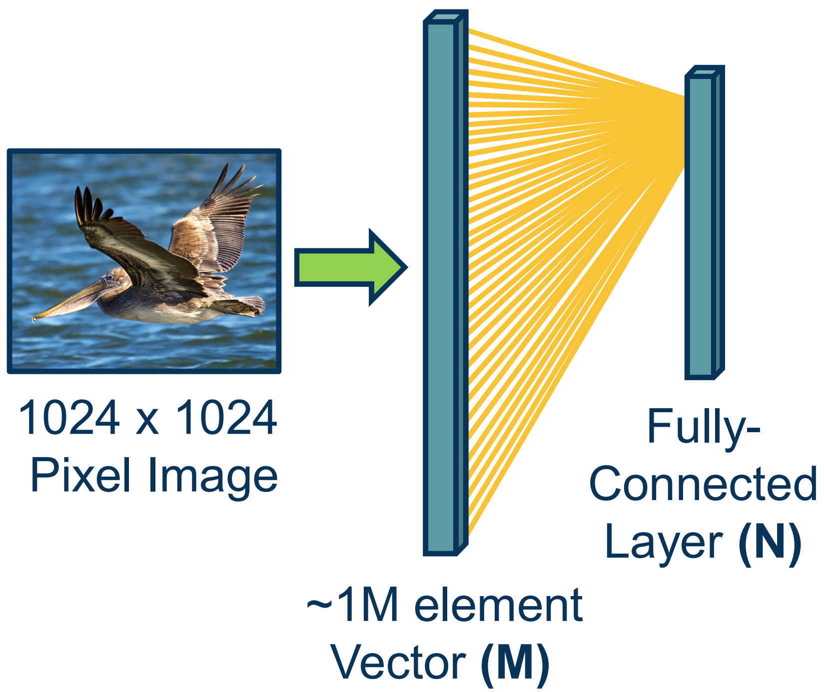
$$\mathrm{CE}(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1-p) & \text{otherwise.} \end{cases}$$

Focal Loss: down-weights easy examples, to give more attention to difficult examples

$$\mathrm{FL}(p_t) = -(1-p_t)^{\gamma} \log(p_t).$$

(Lin et al., 2017)

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?

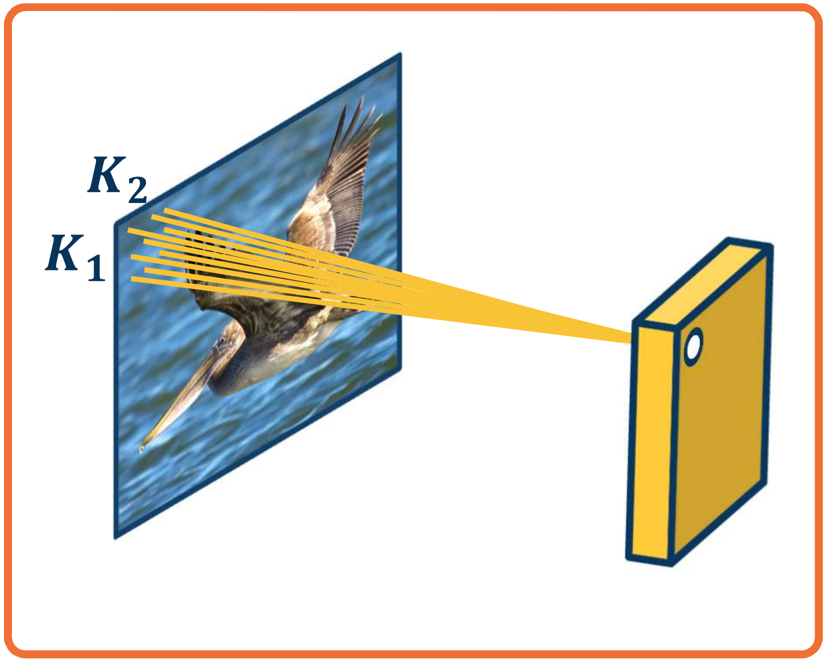
Limitation of Linear Layers

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?

Locality of Features



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

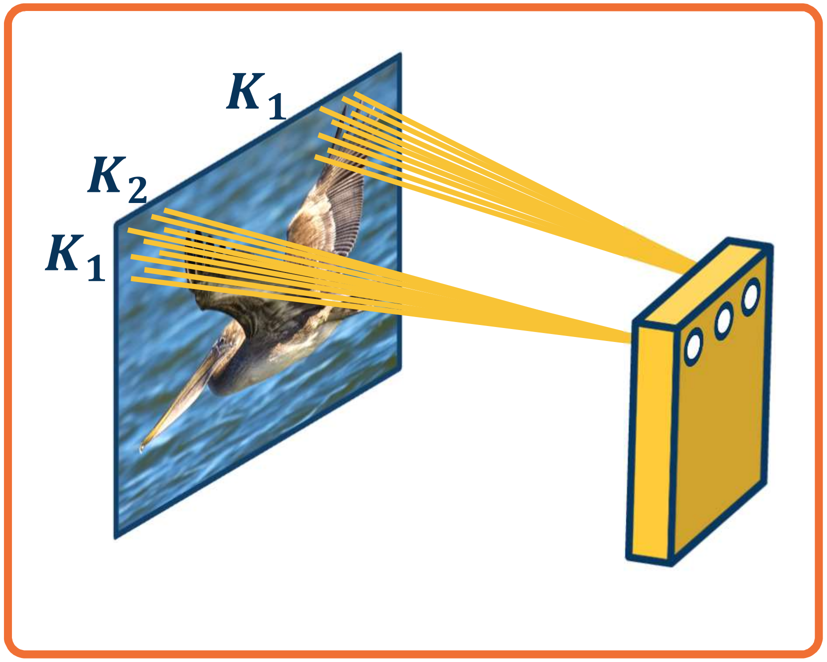
- Region from which a node receives input from is called its **receptive field**

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

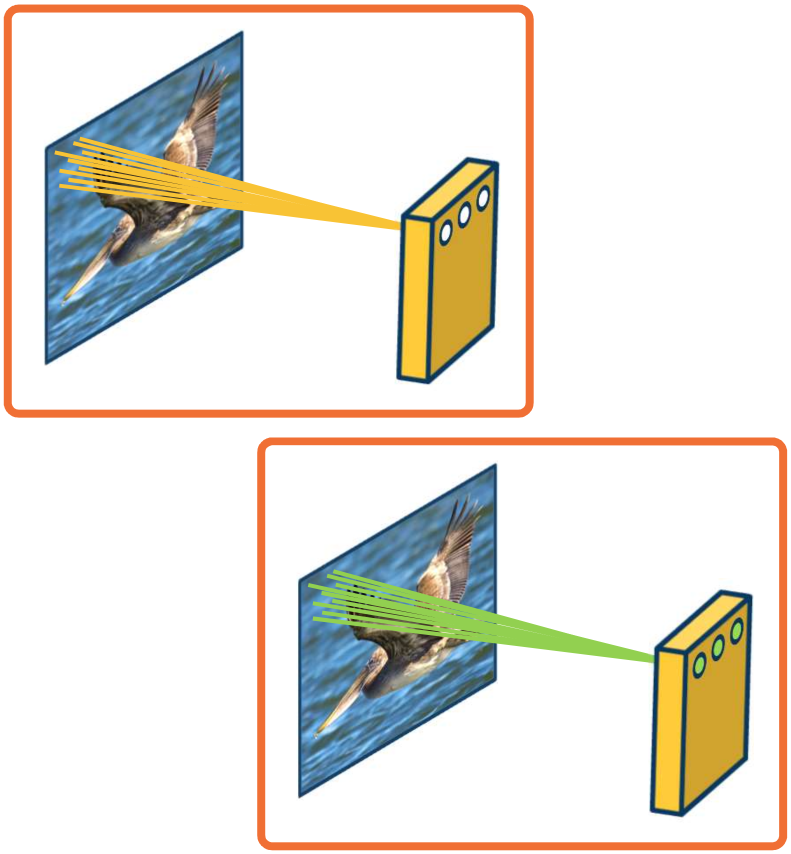
- No reason to think same feature (e.g. edge pattern) can't appear elsewhere
- Use same weights/parameters in computation graph **(shared weights)**

Advantages:

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

Idea 2: Shared Weights

Georgia Tech

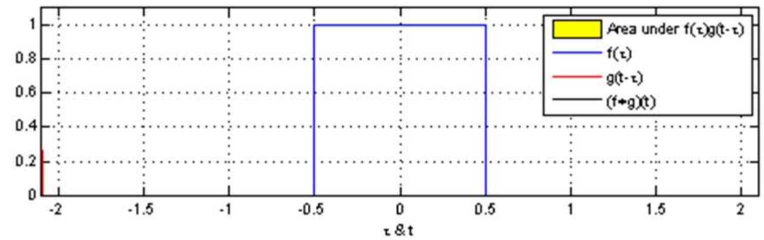


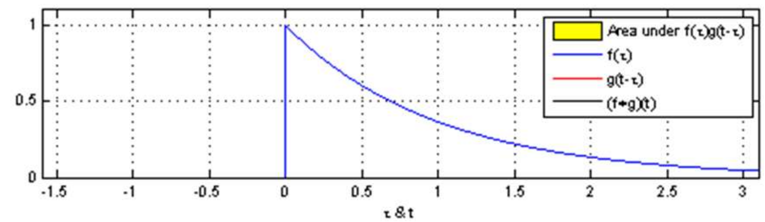
We can learn **many** such features for this one layer

- 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!





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

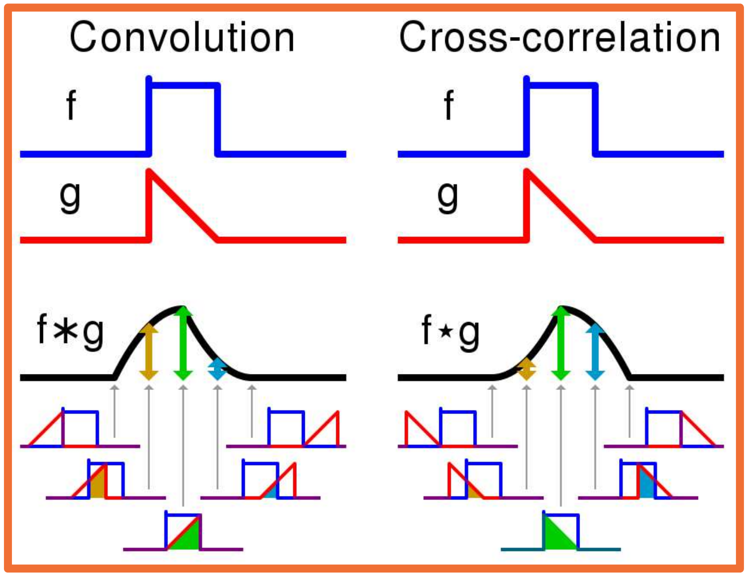
Convolution

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 Discrete Convolution

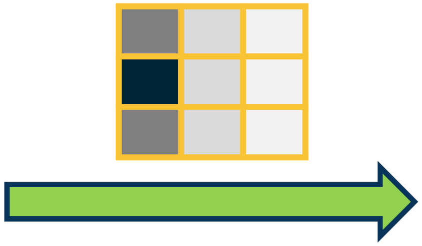
Georgia Tech

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

Georgia Tech

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)

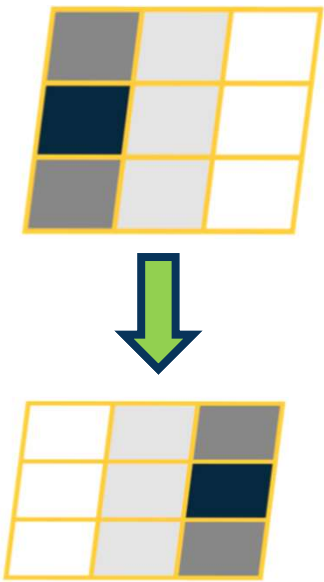
$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Output / filter / feature map

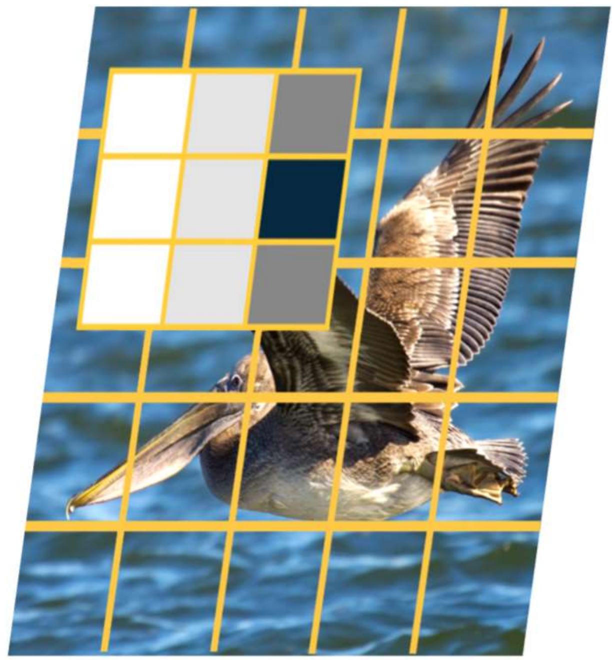
2D Discrete Convolution

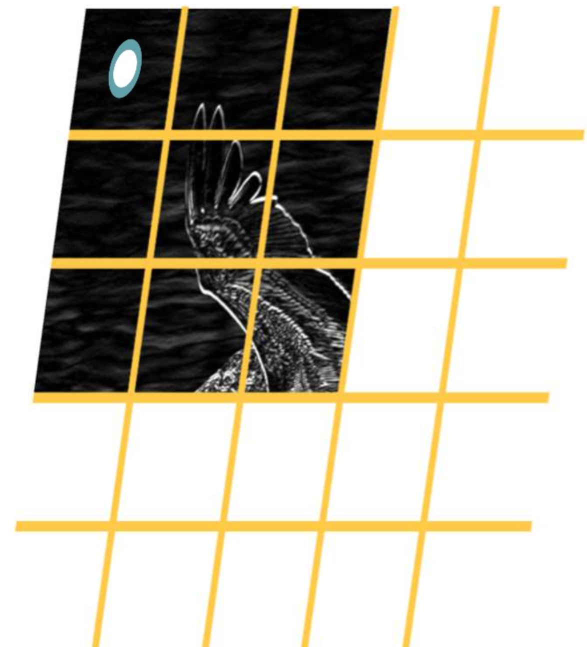
Georgia Tech

1. Flip kernel (rotate 180 degrees)



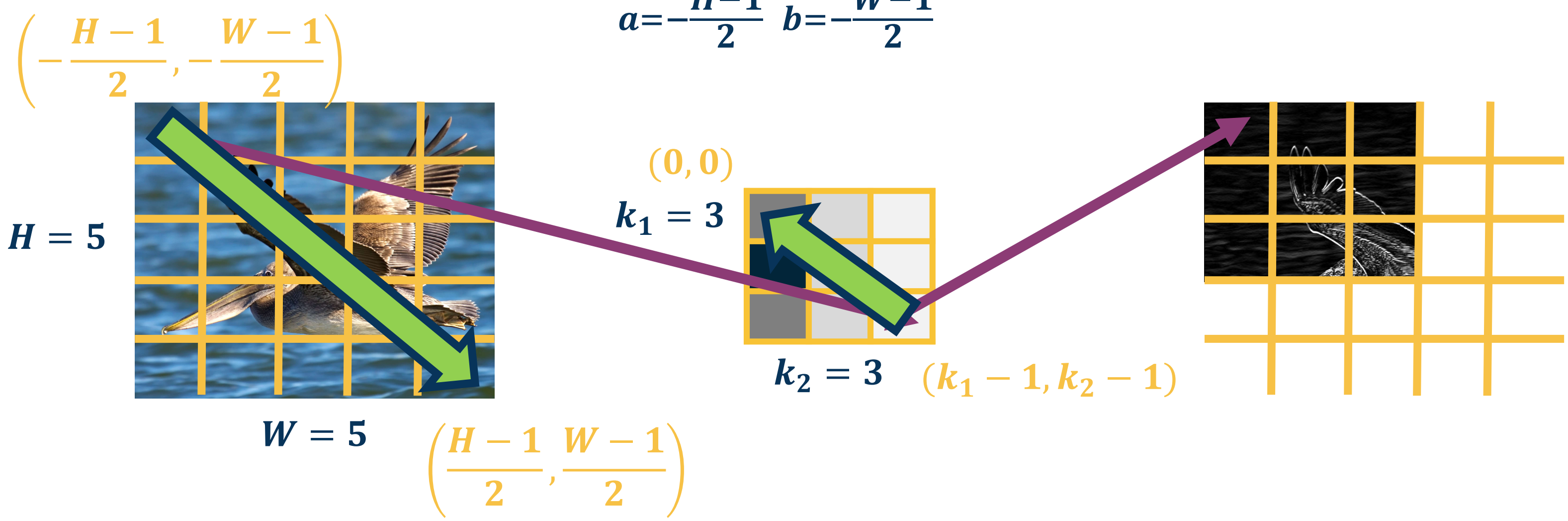
2. Stride along image





The Intuitive Explanation

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



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

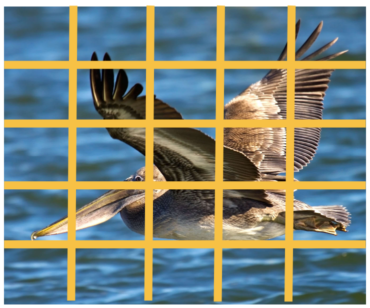
Mathematics of Discrete 2D Convolution

Georgia Tech

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

$(0,0)$

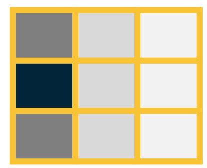
$H = 5$



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

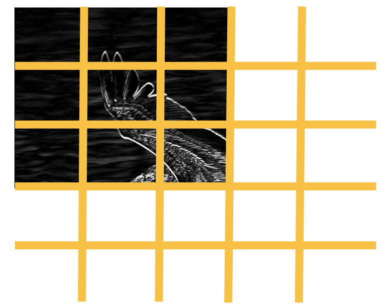
$\left(-\frac{k_1-1}{2}, -\frac{k_2-1}{2}\right)$

$k_1 = 3$



$k_2 = 3$

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



Centering Around the Kernel

Georgia Tech

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)

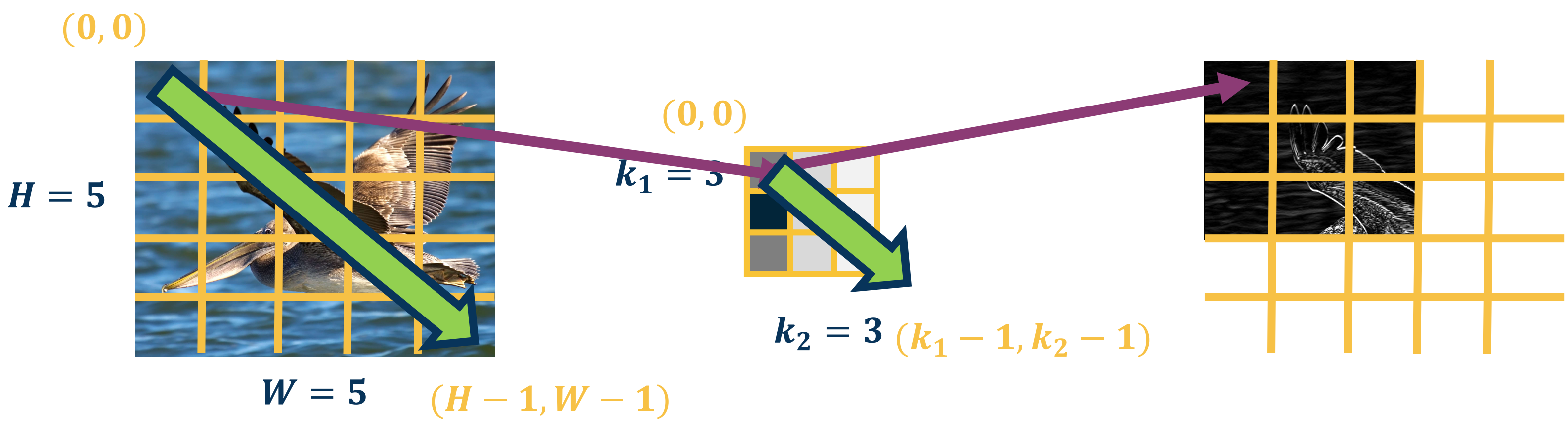
An **intuitive interpretation** of the relationship:

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

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

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

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



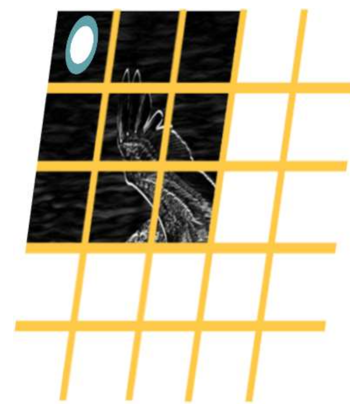
Since we will be learning these kernels, this change does not matter!

Cross-Correlation

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

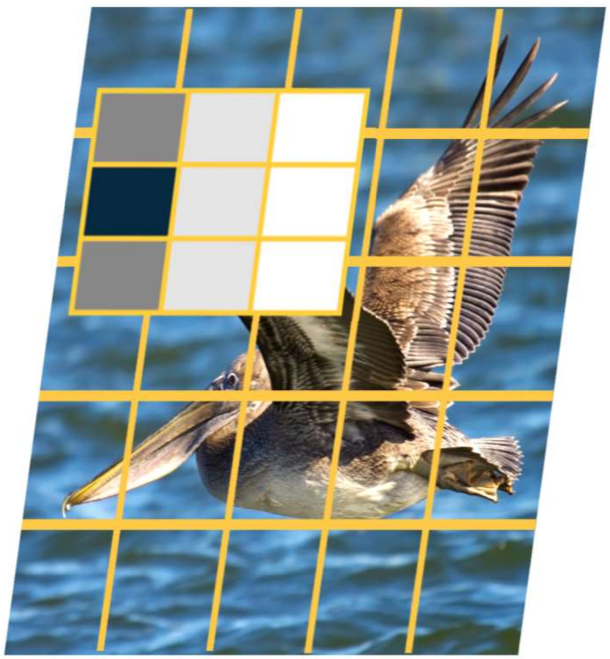
$$X(0:2,0:2) \cdot K' = 65 + \text{bias}$$

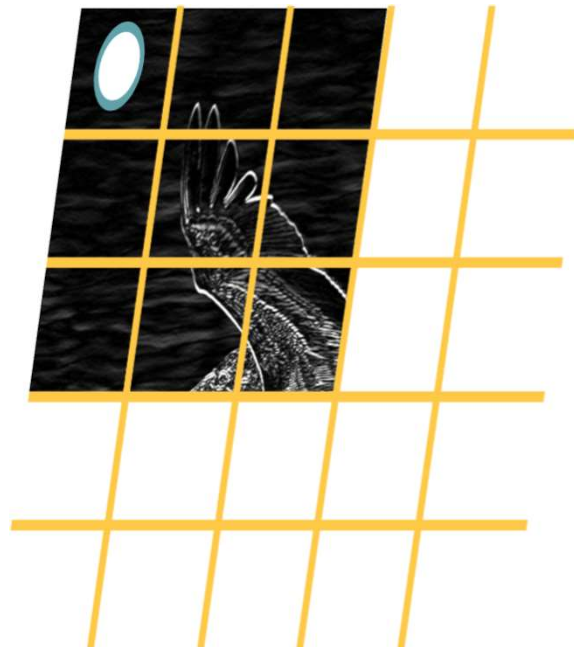
Dot product
(element-wise multiply and sum)



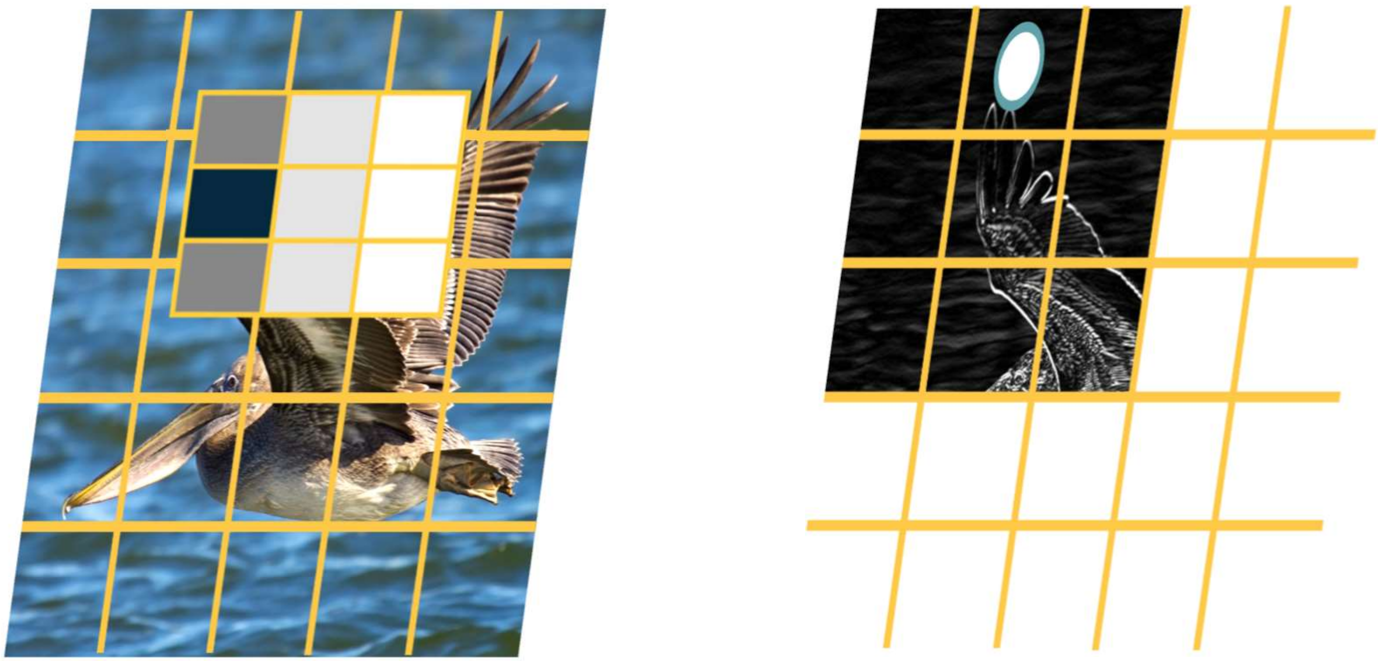
Cross-Correlation

Georgia Tech



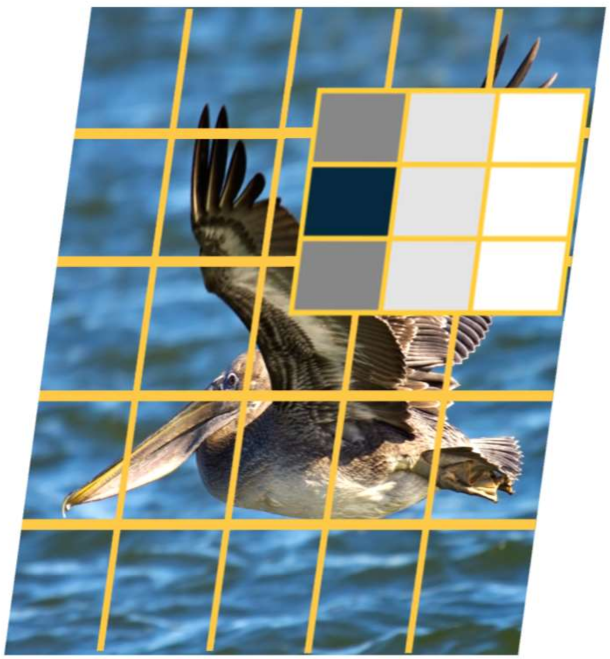


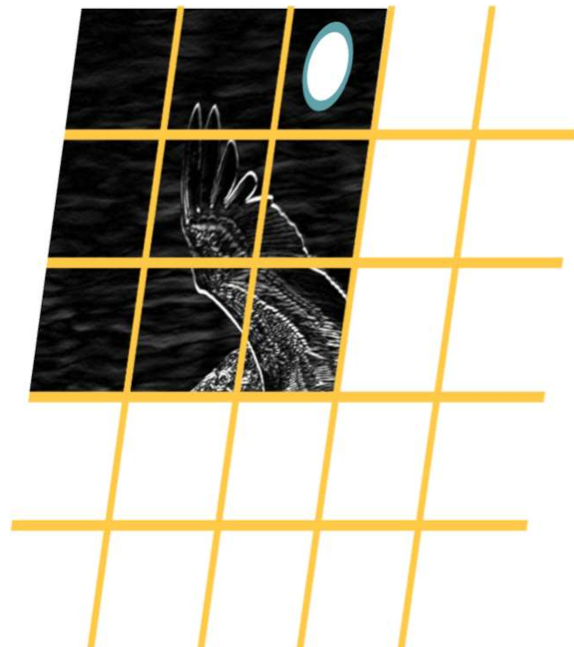
Convolution and Cross-Correlation



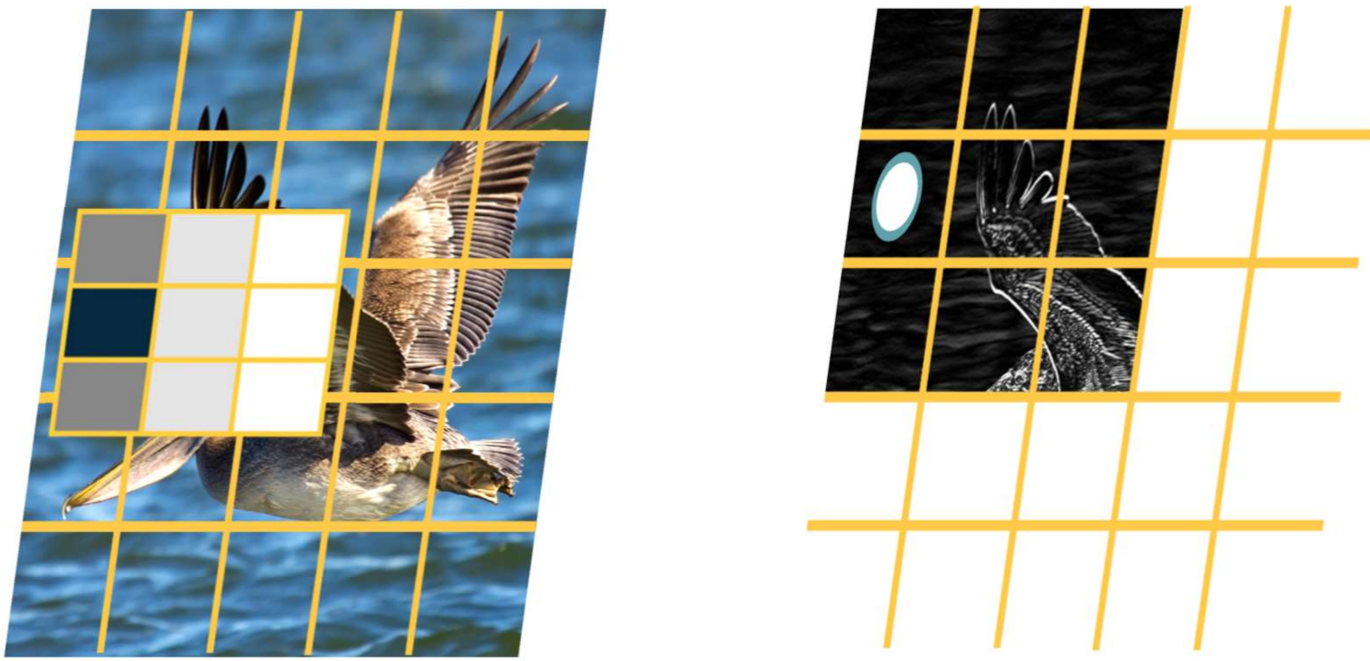
Convolution and Cross-Correlation

Georgia Tech



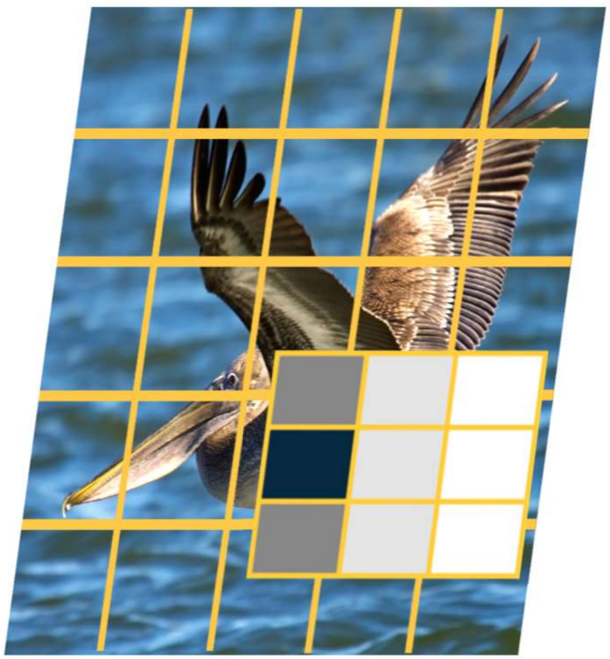


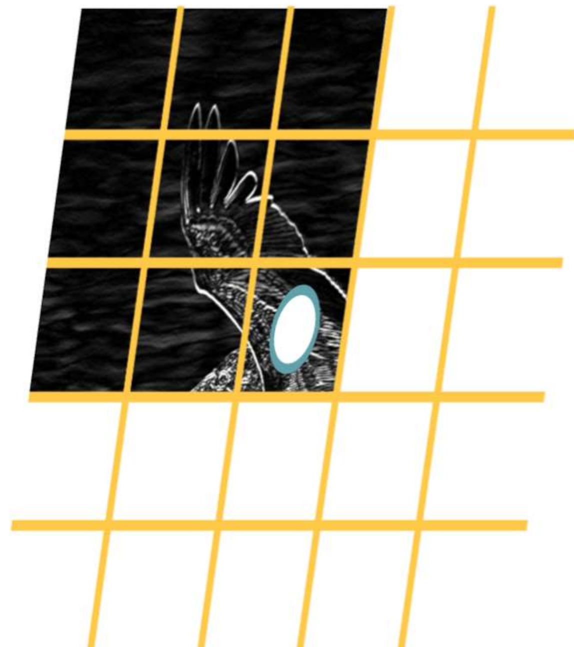
Convolution and Cross-Correlation



Convolution and Cross-Correlation

Georgia Tech





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'`

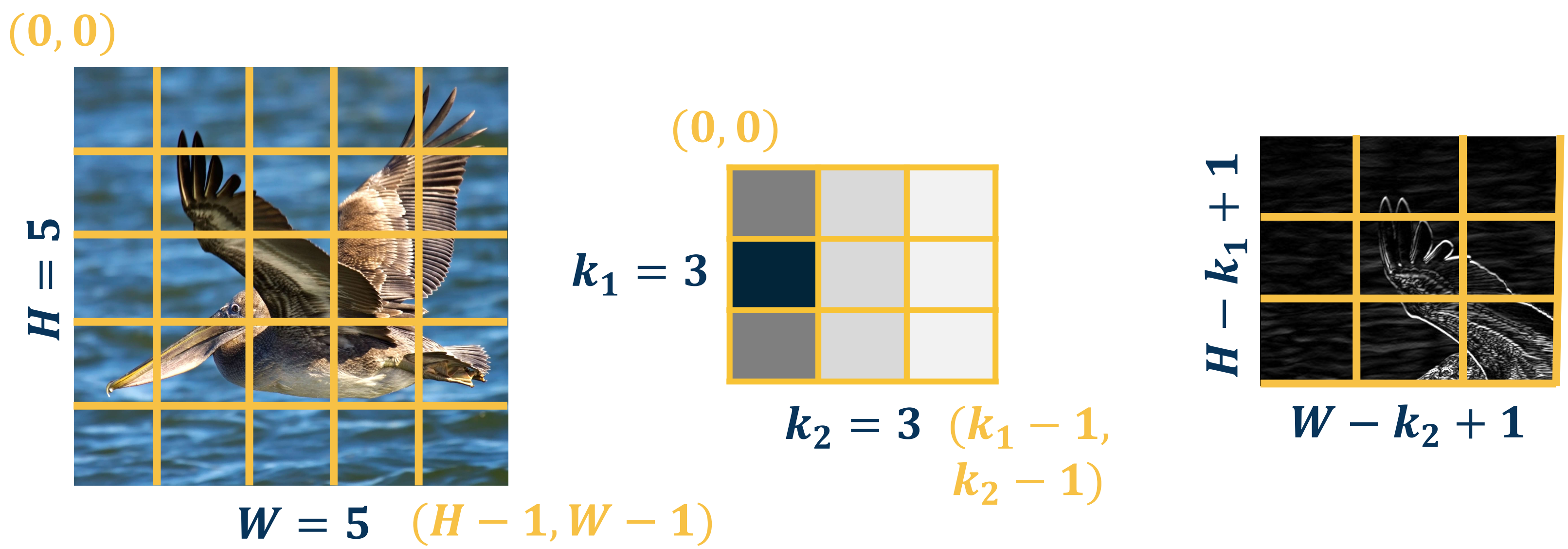
Convolution operations have several hyper-parameters

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

Georgia Tech

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

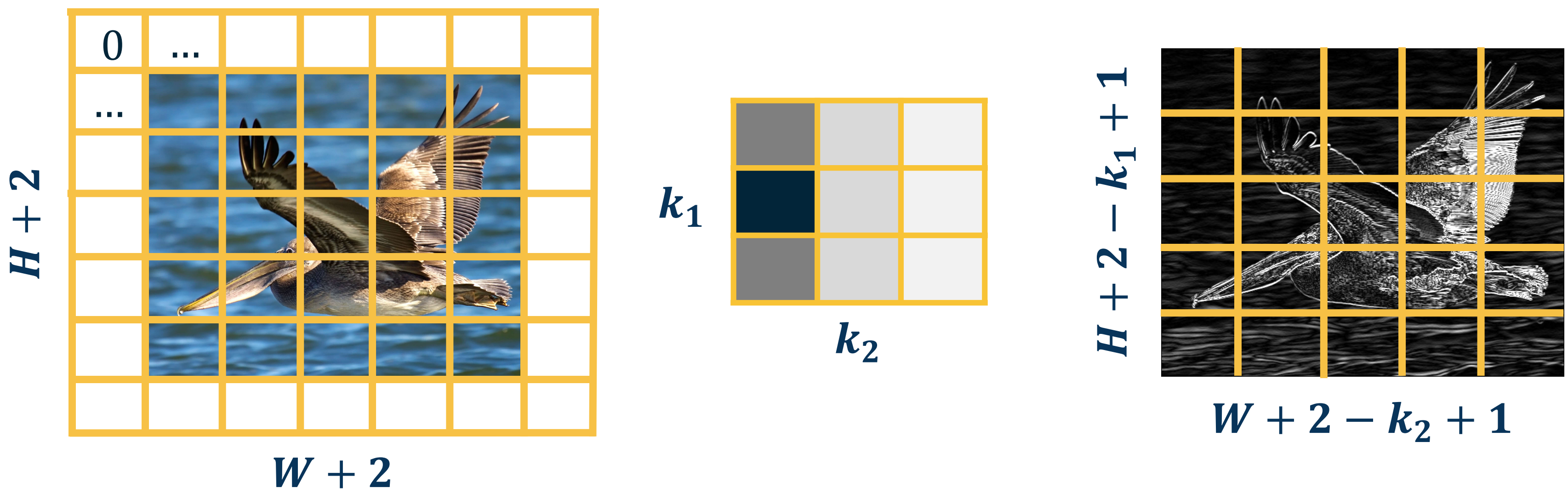


Valid Convolution

Adding Padding

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

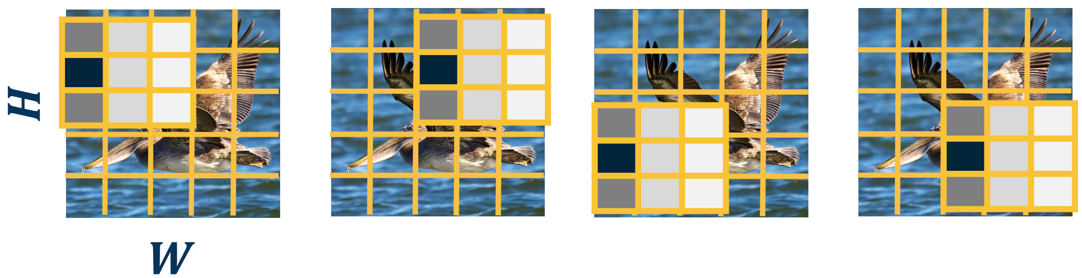
- Zeros, mirrored image, etc.
- Note padding often refers to pixels added to **one size** $(\mathbf{P}=\mathbf{1}\ \text{here})$

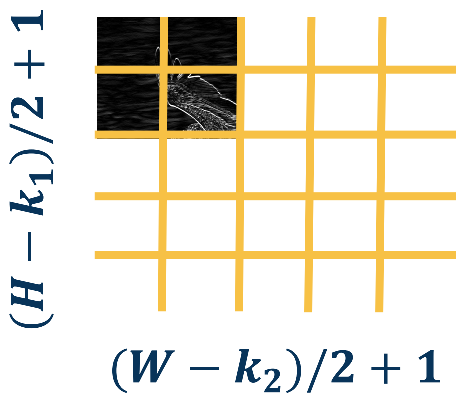


We can move the filter along the image using larger steps **(stride)**

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

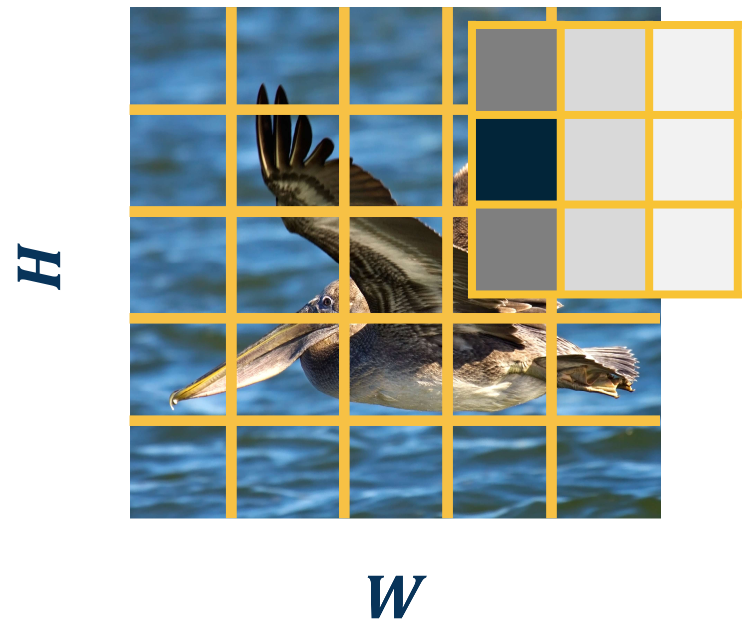
Stride = 2 (every other pixel)





Stride

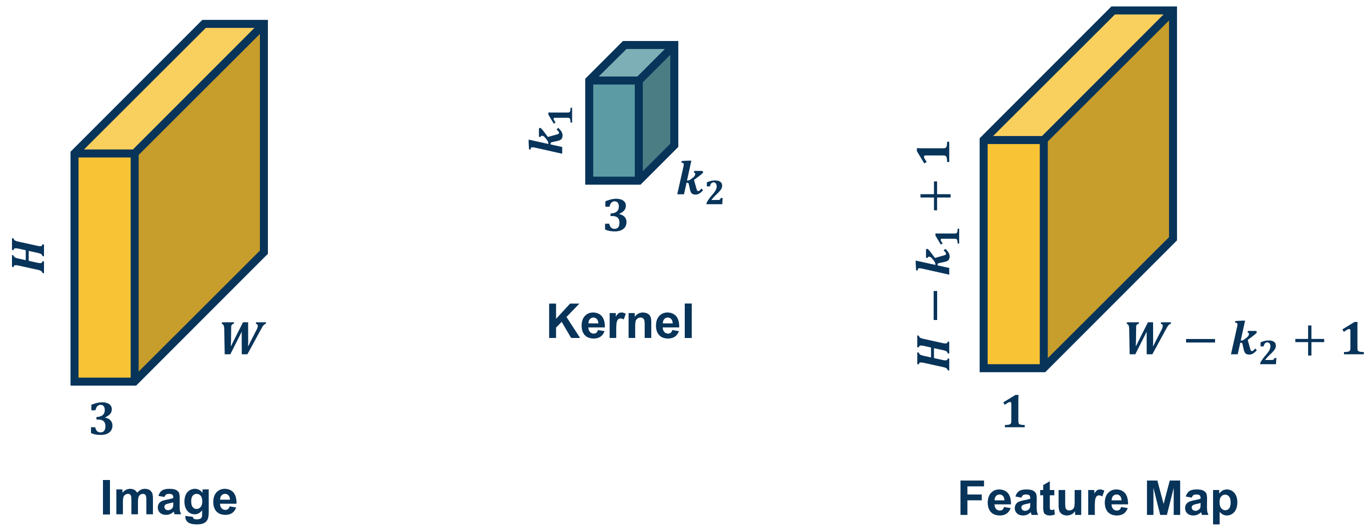
Stride can result in **skipped pixels**, e.g. stride of 3 for 5x5 input



Invalid Stride

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

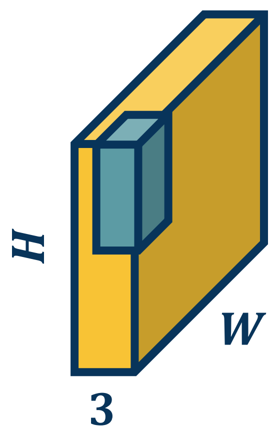
- In such cases, we have **3-channel kernels**!



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

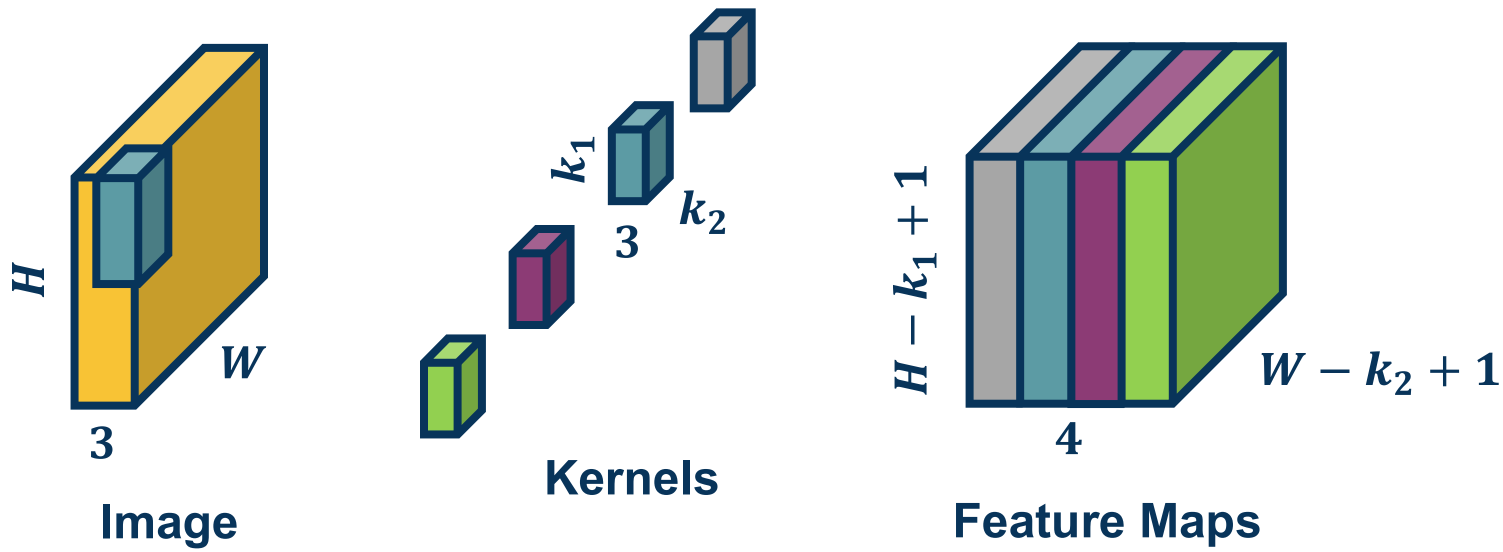
Similar to before, we perform **element-wise multiplication** between kernel and image patch, summing them up **(dot product)**

- 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*

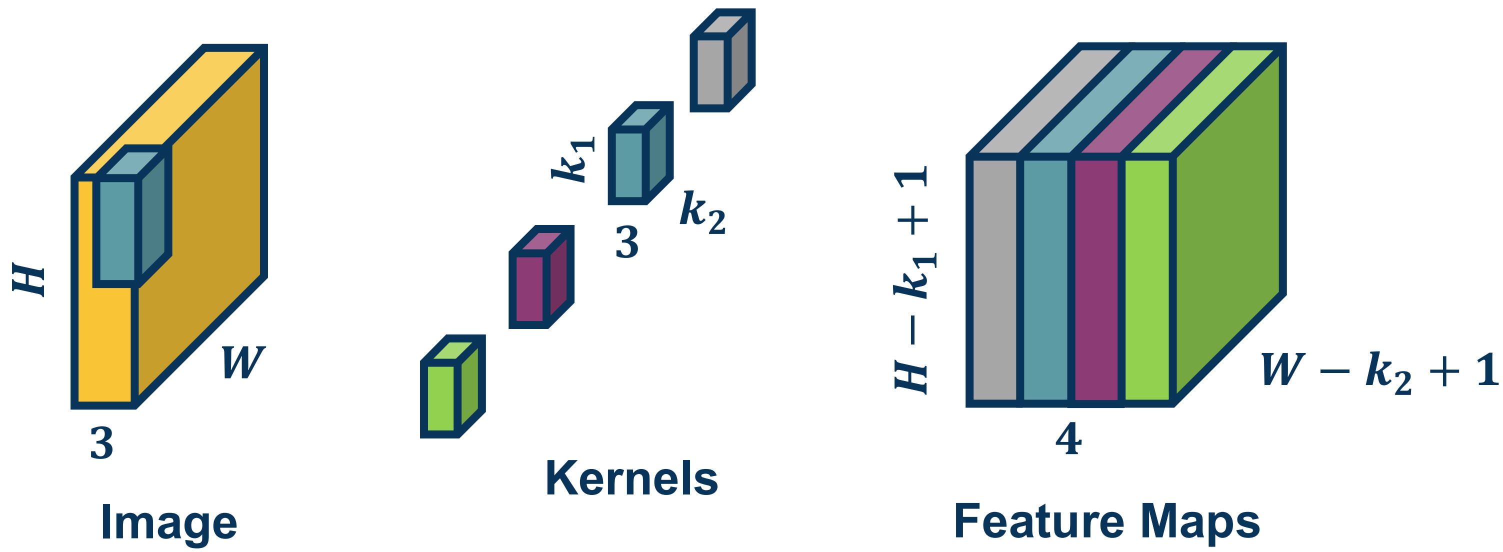


Multiple Kernels

Georgia Tech

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

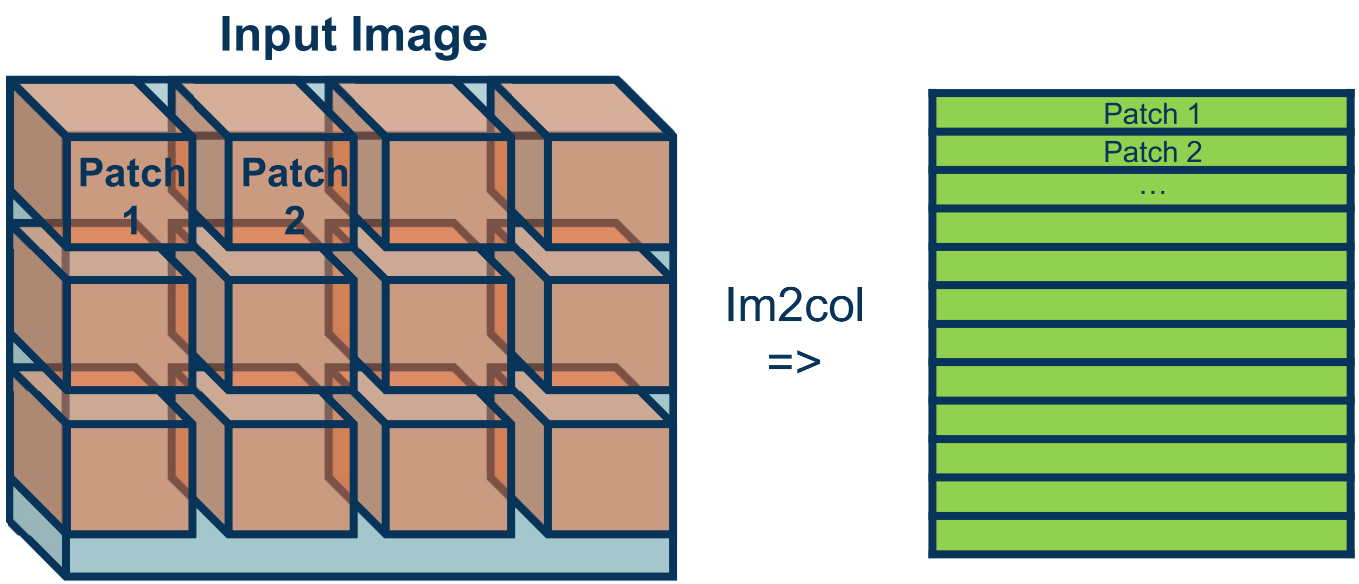
- Example:
  $k_1 = 3, k_2 = 3,\ N = 4\ input\ channels = 3$, then $(3 * 3 * 3 + 1) * 4 =$112



Number of Parameters

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

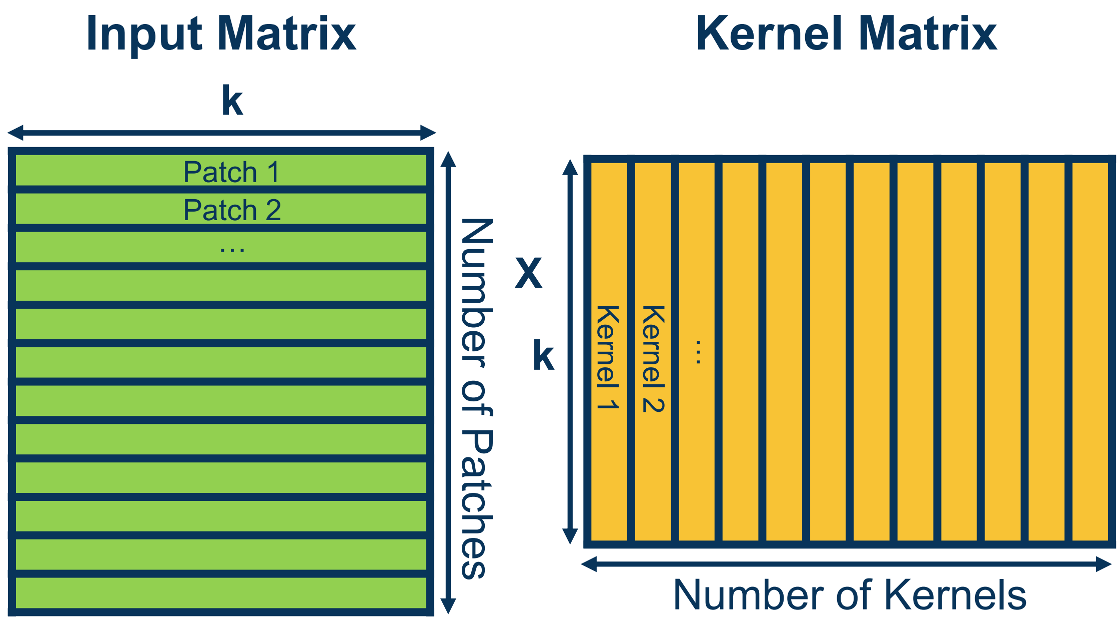
- **Step 1:** Lay out image patches in vector form (note can overlap!)



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

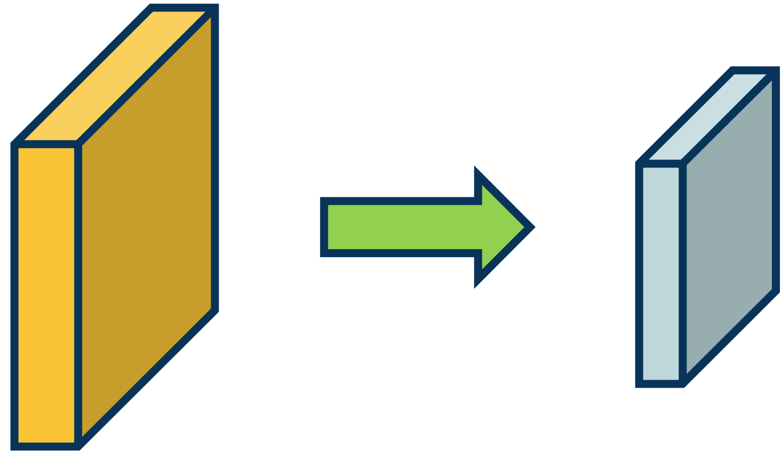
- **Step 2**: Multiple patches by kernels



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

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

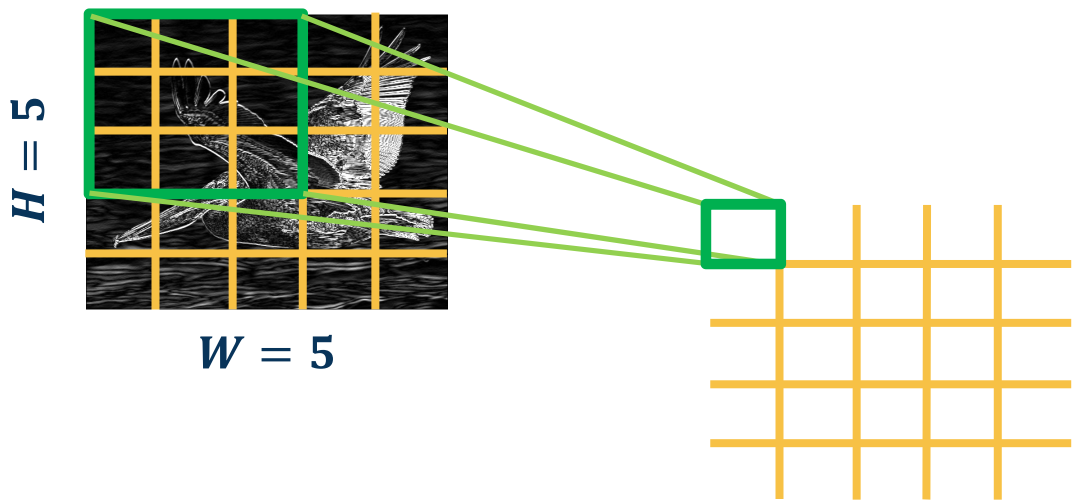
Pooling Layers

Georgia Tech

Example: Max pooling

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

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



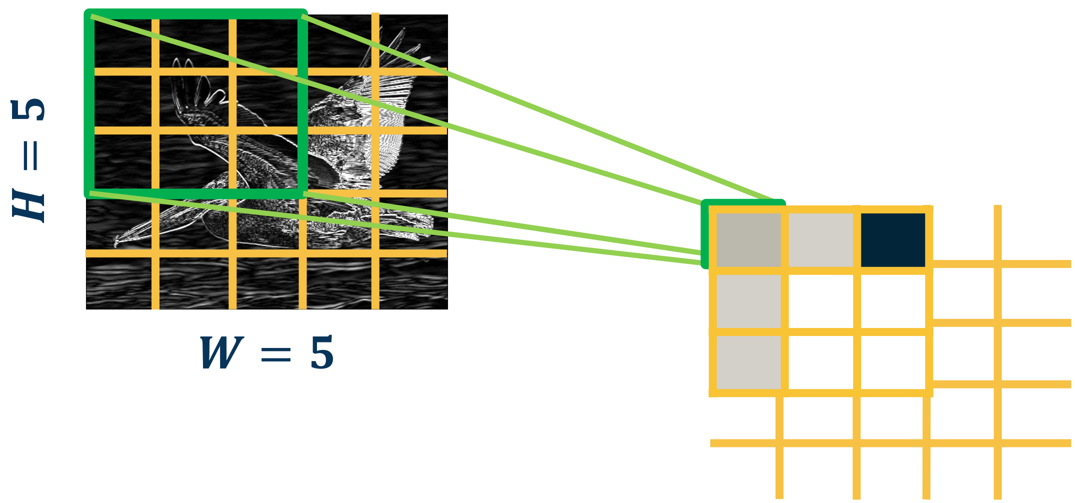
How many learned parameters does this layer have?

None!

Not restricted to max; can use any differentiable function

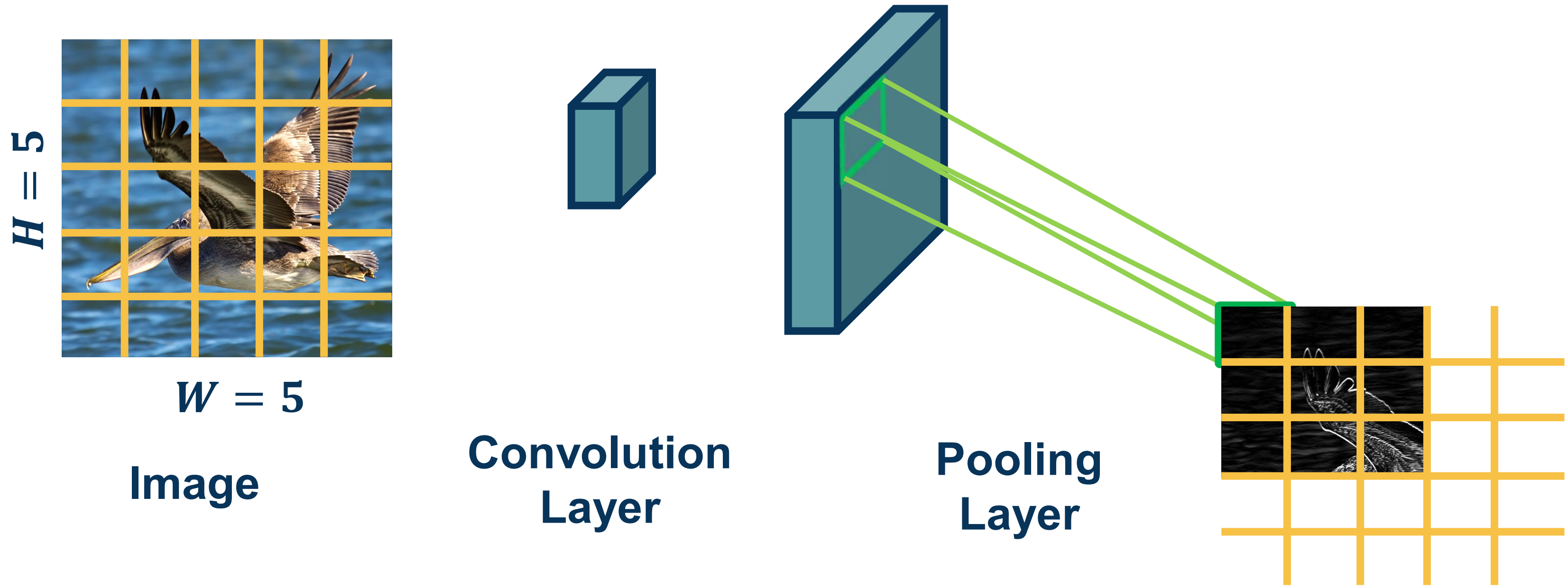
- Not very common in practice

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



Max Pooling

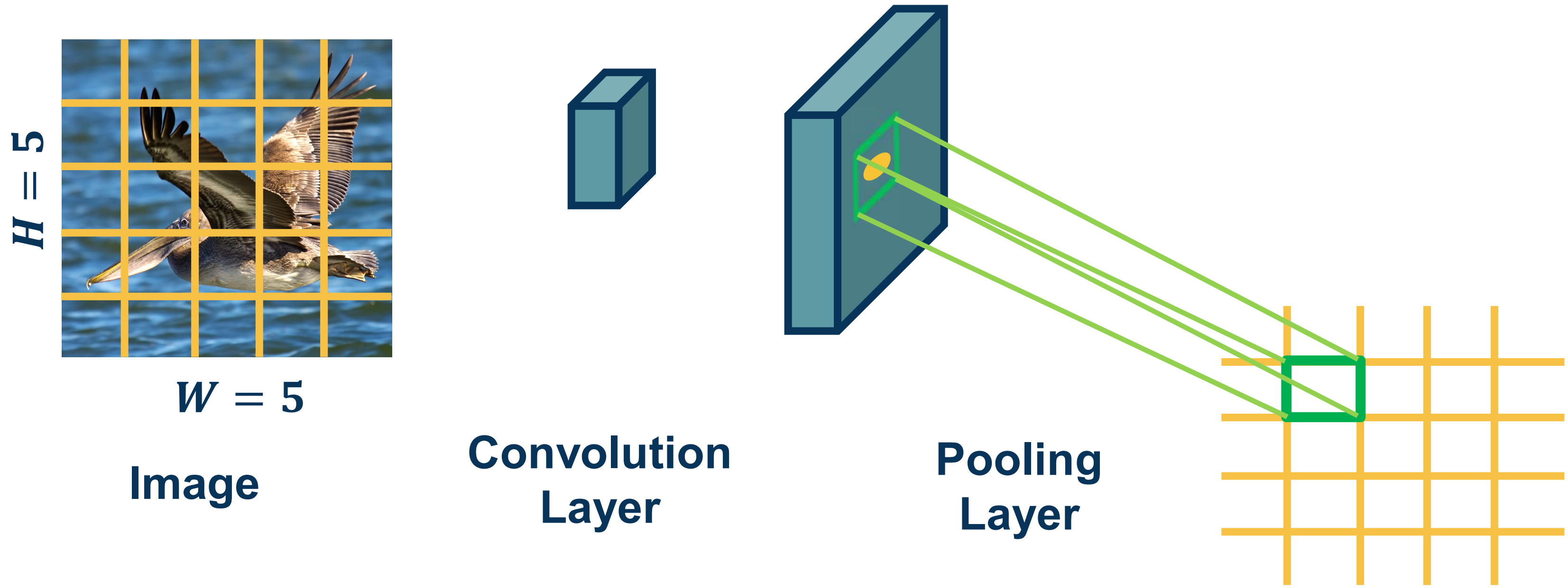
Since the **output** of convolution and pooling layers are **(multi-channel) images**, we can sequence them just as any other layer



Combining Convolution & Pooling Layers

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

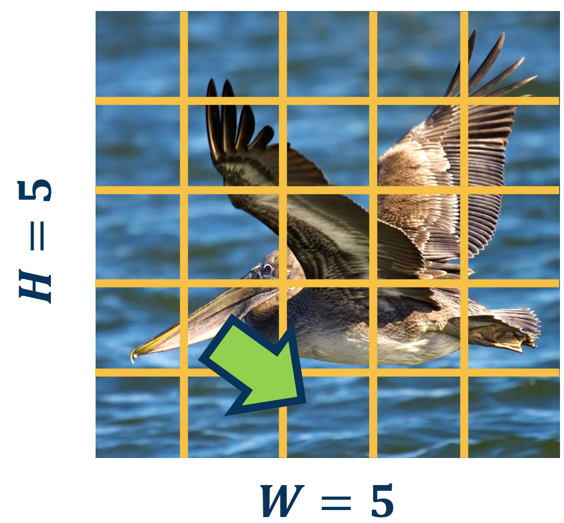
- If feature (such as beak) translated a little bit, output values still **remain the same**

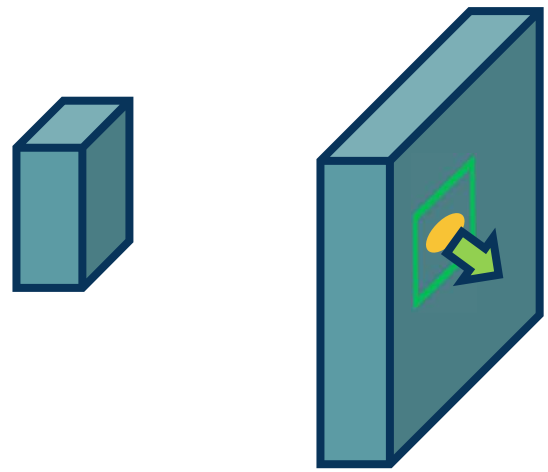


Invariance

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

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





Invariance vs. Equivariance

Georgia Tech

Backwards Pass for Convolution Layer

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

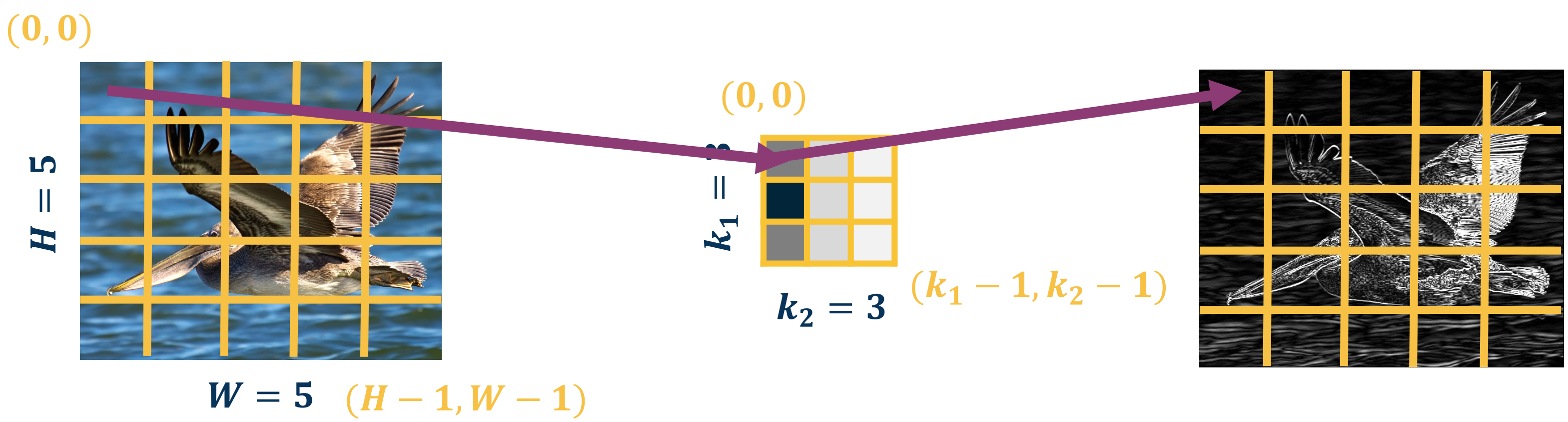
- Similar to fully connected layer, will be **simple vectorized linear algebra operation**!
- We will see a **duality** between cross-correlation and convolution

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

$$\Downarrow$$

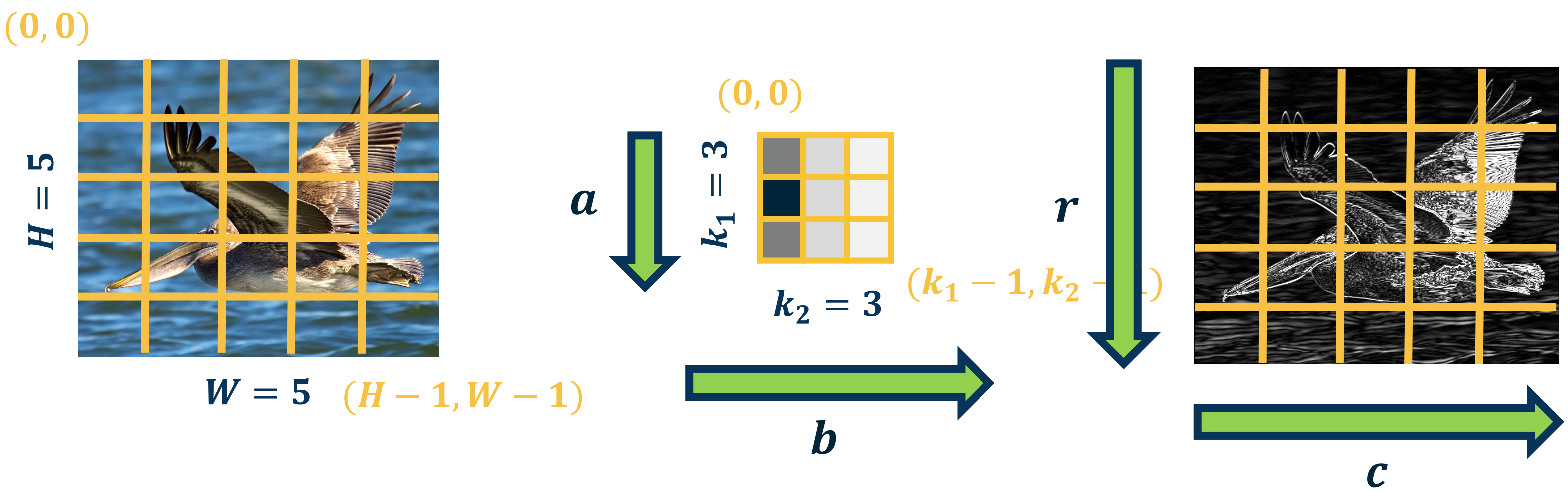
$$K' = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

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



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

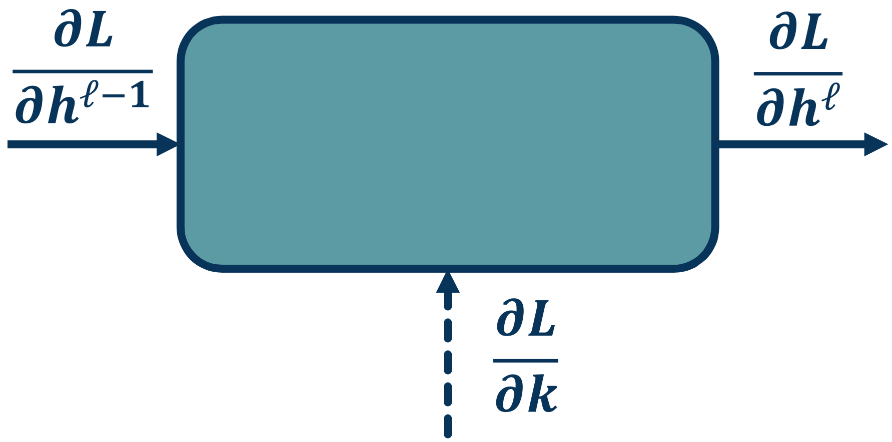
$$y(r,c) = (x * k)(r,c) = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} x(r+a, c+b)\, k(a,b)$$

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

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



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

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

Georgia Tech

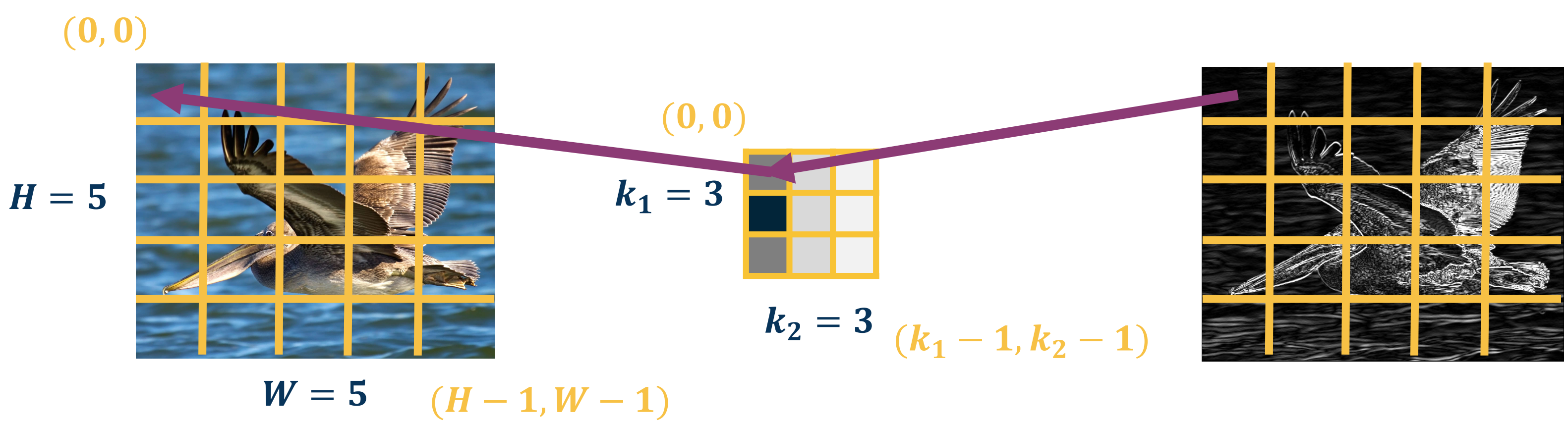
$$\frac{\partial L}{\partial k} = \frac{\partial L}{\partial h^{\ell}} \frac{\partial h^{\ell}}{\partial k}$$

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

Georgia Tech

Chain Rule over all Output Pixels

Need to incorporate all upstream gradients:

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

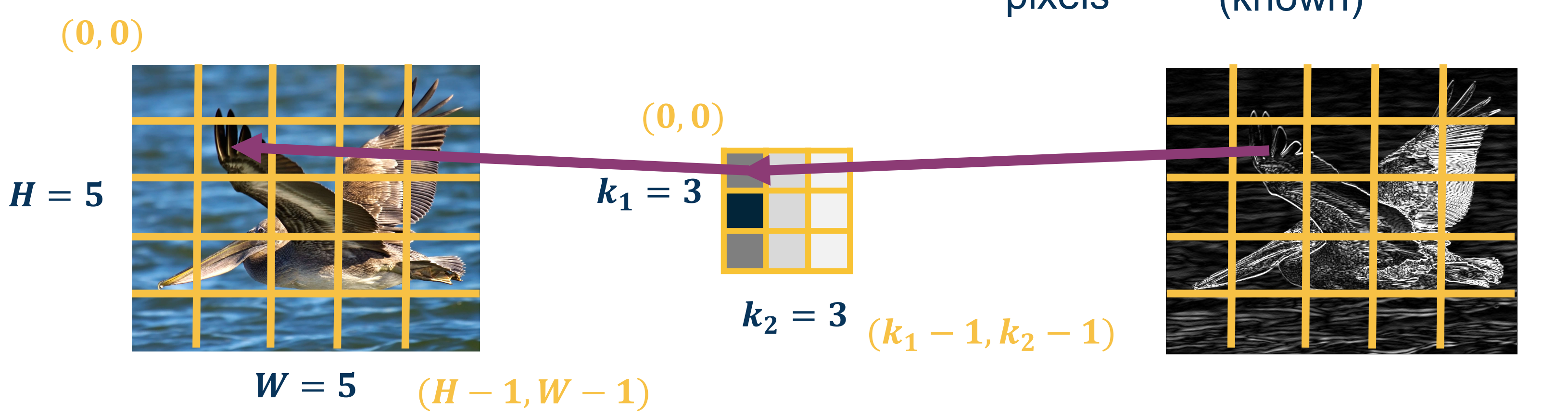
Chain Rule:

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

Sum over all output pixels

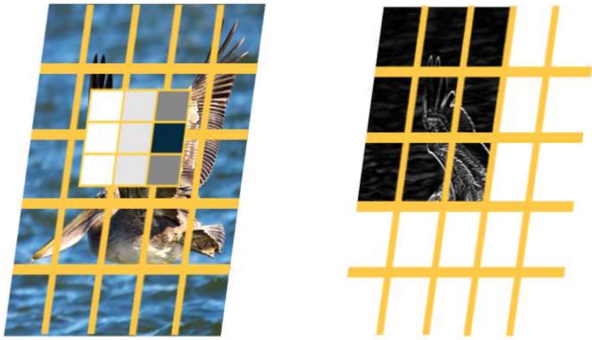
Upstream gradient (known)

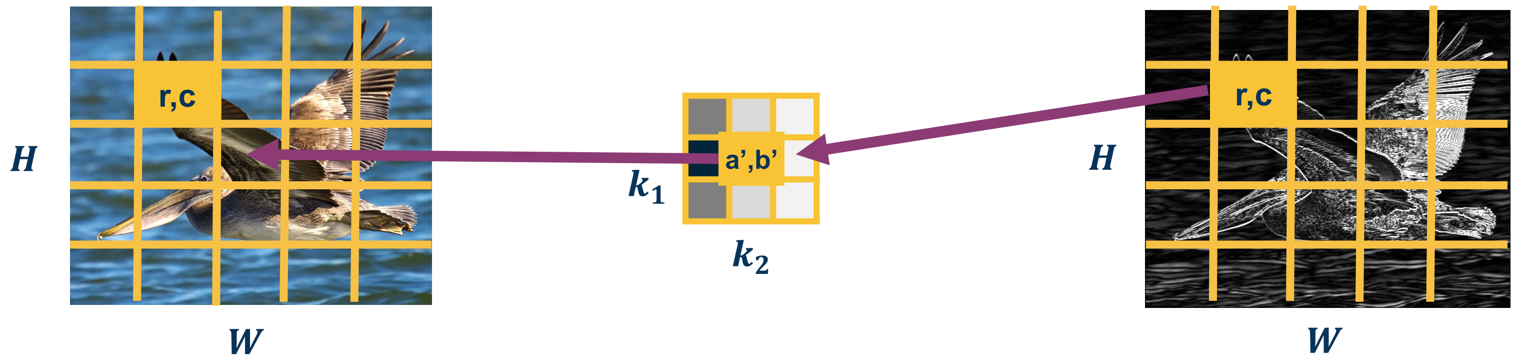
We will compute



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

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





Chain Rule over all Output Pixels

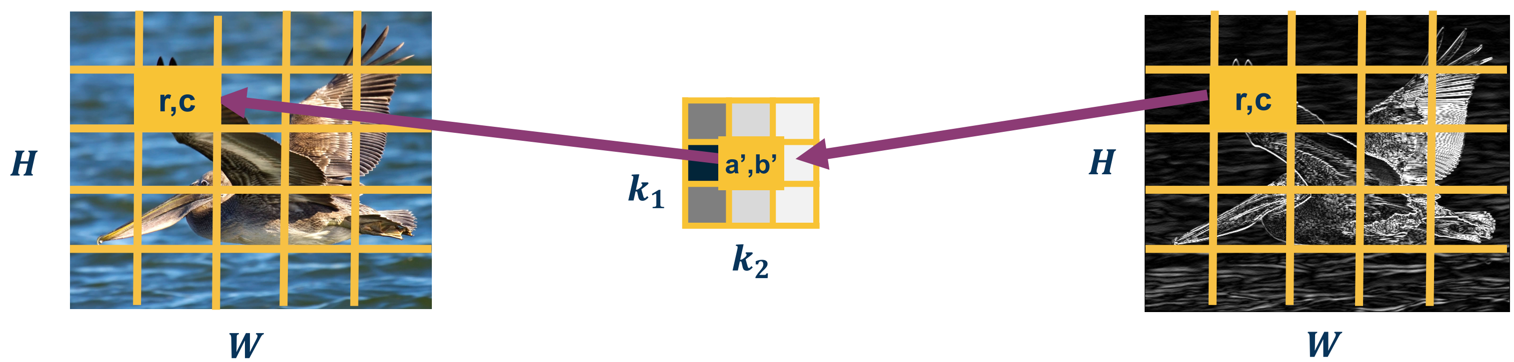
$$\frac{\partial y(r,c)}{\partial k(a',b')} = x(r+a', c+b')$$

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

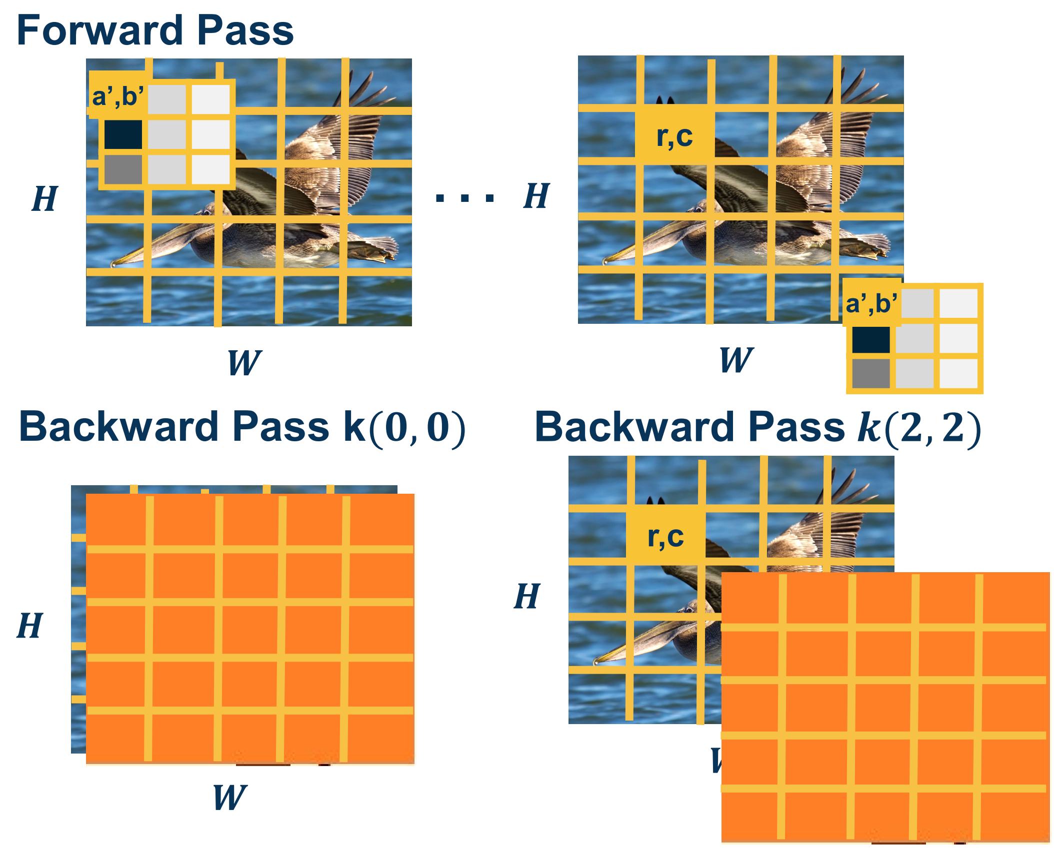
Does this look familiar?

Cross-correlation between upstream gradient and input!

(until $k_1 \times k_2$ output)



Gradients and Cross-Correlation



Does this look familiar?

Cross-correlation between upstream gradient and input!

(until $k_1 \times k_2$ output)

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

Gradient for input (to pass to prior layer)

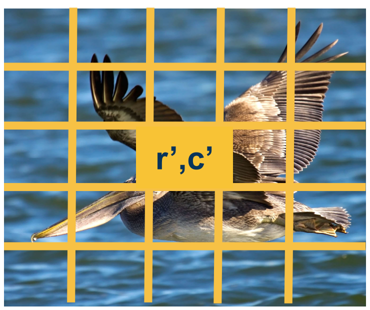
Calculate one pixel at a time $\frac{\partial L}{\partial x(r', c')}$

What does this input pixel affect at the output?

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

$(0,0)$

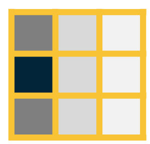
$H = 5$



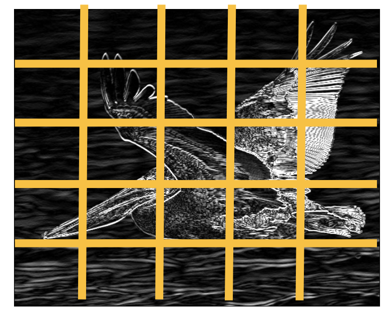
$W = 5$ $(H-1, W-1)$

$(0,0)$

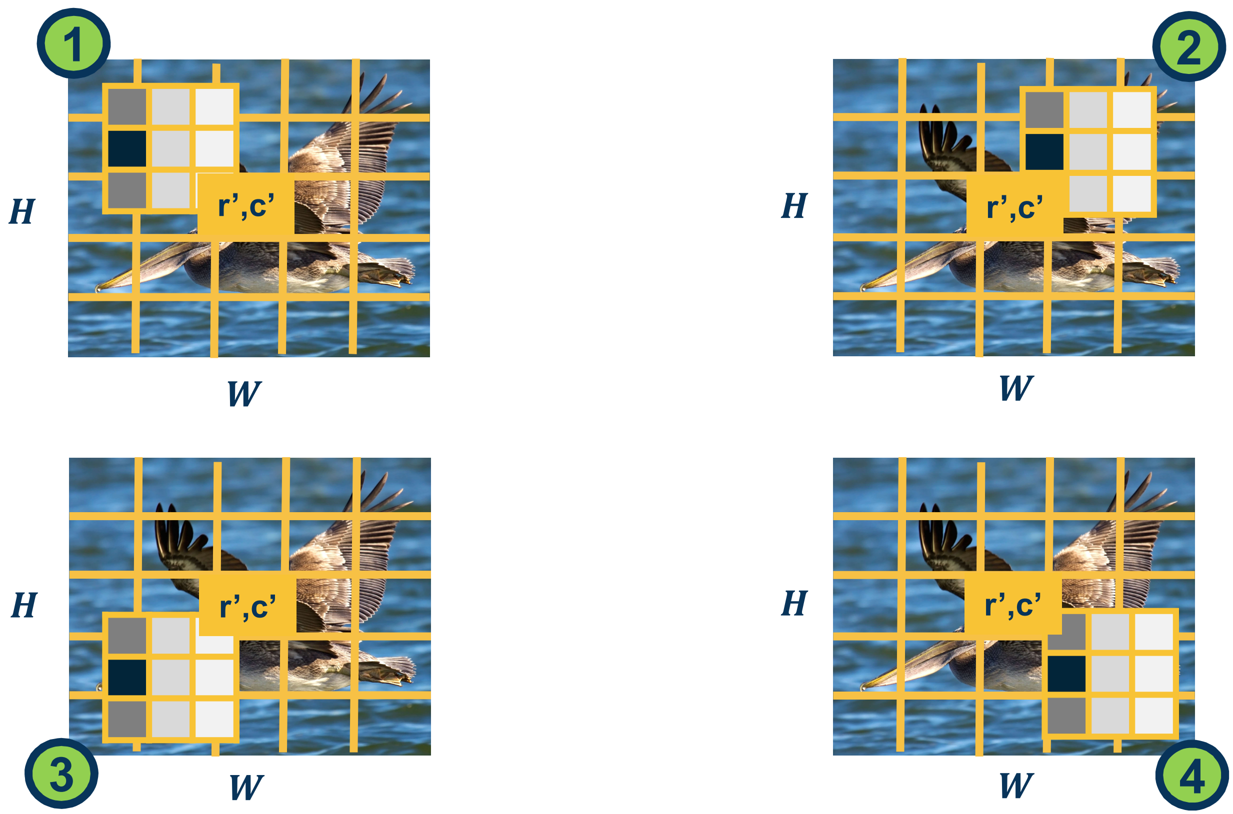
$k_1 = 3$



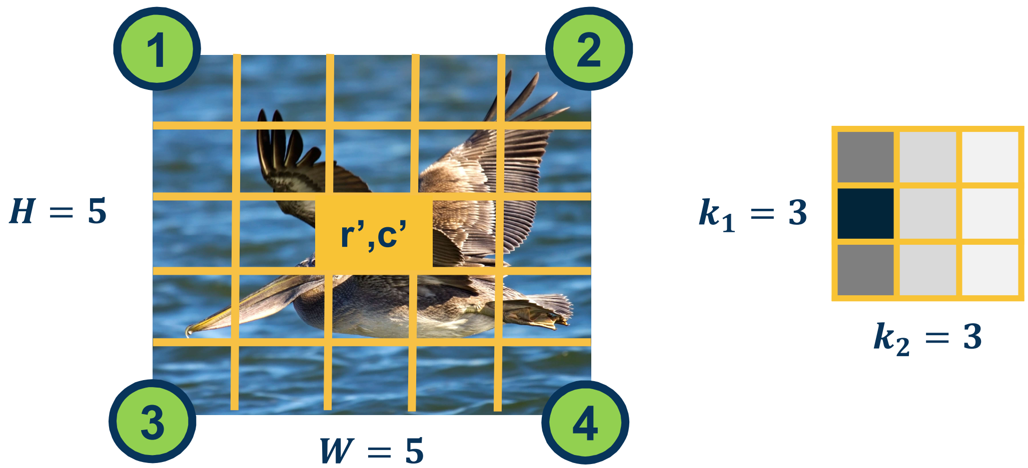
$k_2 = 3$ $(k_1 - 1, k_2 - 1)$

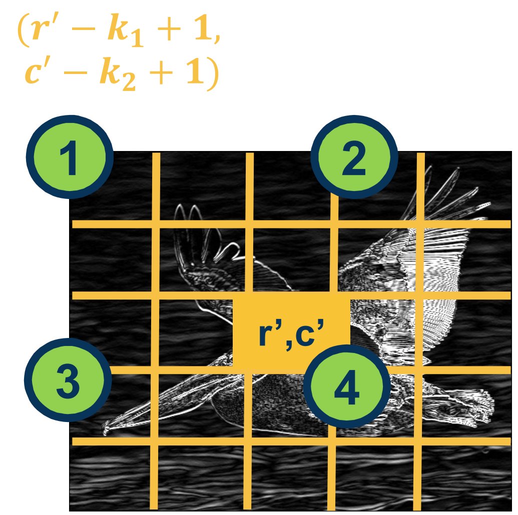


What an Input Pixel Affects at Output



Extents of Kernel Touching the Pixel





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

Extents at the Output

Georgia Tech

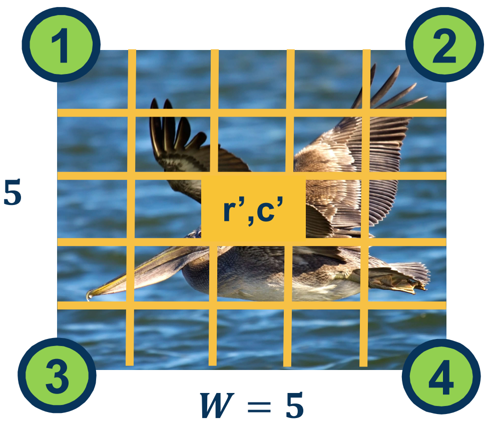
Chain rule for affected pixels (sum gradients):

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

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

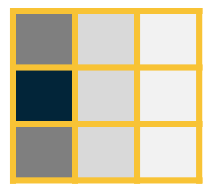
Let's derive it analytically this time (as opposed to visually)

$H = 5$



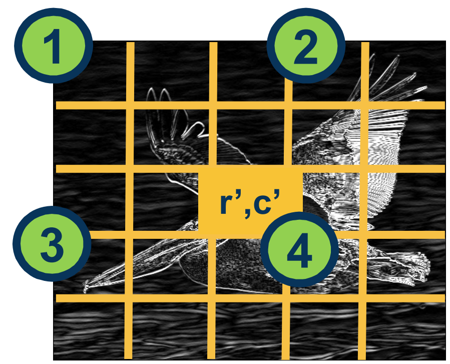
$W = 5$

$k_1 = 3$



$k_2 = 3$

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



Summing Gradient Contributions

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

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

Plug in what we actually wanted :

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

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 $\mathbf{b} = \mathbf{b}'$

Calculating the Gradient

Georgia Tech

Plugging in to earlier equation:

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

Does this look familiar?

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

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