Topics:

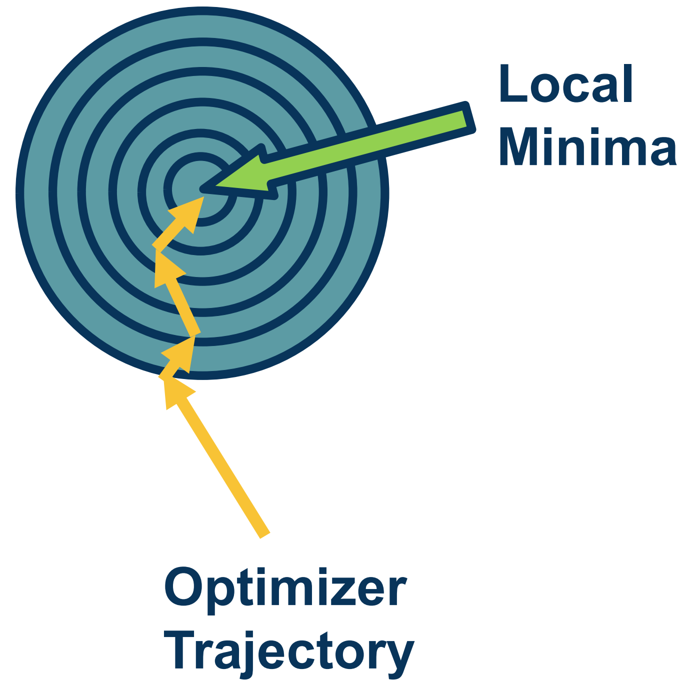
- Convolution

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
    - Backprop notes https://www.cc.gatech.edu/classes/AY2022/cs7643_spring/assets/L10_cnns_backprop_notes.pdf
    - There will be various OH tutorials
    - Slower OMSCS lectures on dropbox: Module 2 Lessons 5-6 (M2L5/M2L6) (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**

Even given a good neural network architecture, we need a **good optimization algorithm to find good weights**

- What **optimizer** should we use?
  - Different optimizers make **different weight updates** depending on the gradients
- How should we **initialize** the weights?
- What **regularizers** should we use?
- What **loss function** is appropriate?



Optimization Considerations

Key idea: Rather than combining velocity with current gradient, go along velocity **first** and then calculate gradient at new point

- We know velocity is probably a **reasonable direction**

$$\hat{w}_{i-1} = w_{i-1} + \beta v_{i-1}$$

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial \hat{w}_{i-1}}$$

$$w_i = w_{i-1} - \alpha v_i$$

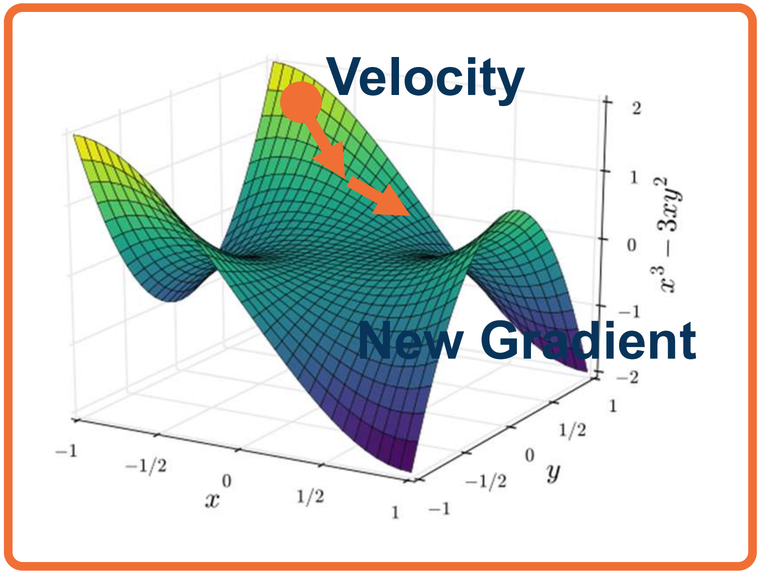


Figure Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Solution: Time-varying bias correction

Typically $\beta_1 = 0.9, \ \beta_2 = 0.999$

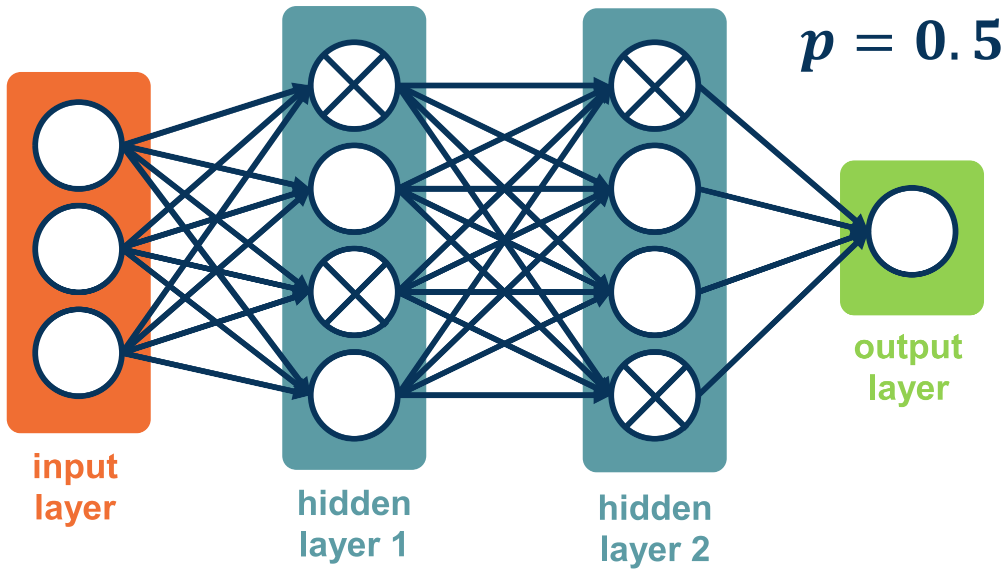
So $\hat{v}_i$ will be small number divided by (1-0.9=0.1) resulting in more reasonable values (and $\hat{G}_i$ larger)

$$v_i = \beta_1 \, v_{i-1} + (1 - \beta_1)\left(\frac{\partial L}{\partial w_{i-1}}\right)$$

$$G_i = \beta_2 \, G_{i-1} + (1 - \beta_2)\left(\frac{\partial L}{\partial w_{i-1}}\right)^2$$

$$\hat{v}_i = \frac{v_i}{1 - \beta_1^t} \qquad \hat{G}_i = \frac{G_i}{1 - \beta_2^t}$$

$$w_i = w_{i-1} - \frac{\alpha \, \hat{v}_i}{\sqrt{\hat{G}_i + \epsilon}}$$



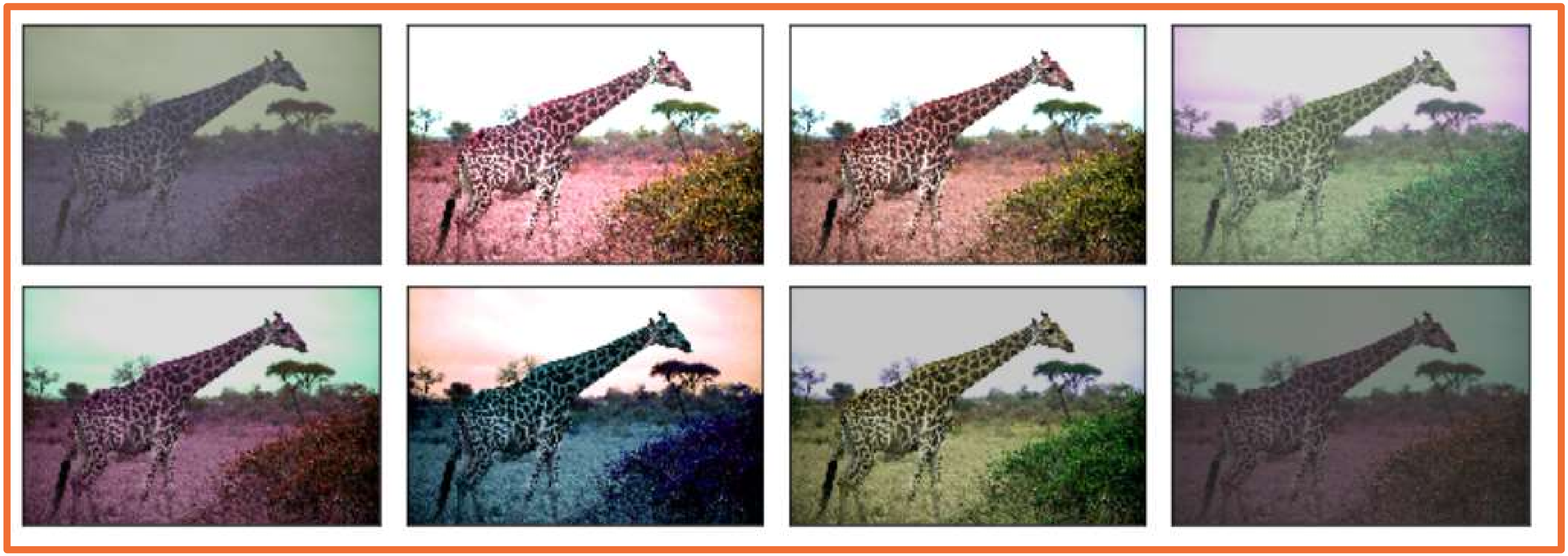
An idea: For each node, keep its output with probability p

- Activations of deactivated nodes are essentially zero

Choose whether to mask out a particular node **each iteration**

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

Color Jitter



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

- We can give the model flexibility through **learnable parameters γ (scale) and β (shift)**
- Network can learn to **not normalize** if necessary!
- This layer is called a **Batch Normalization (BN) layer**

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

From: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Sergey Ioffe, Christian Szegedy

Learnable Scaling and Offset

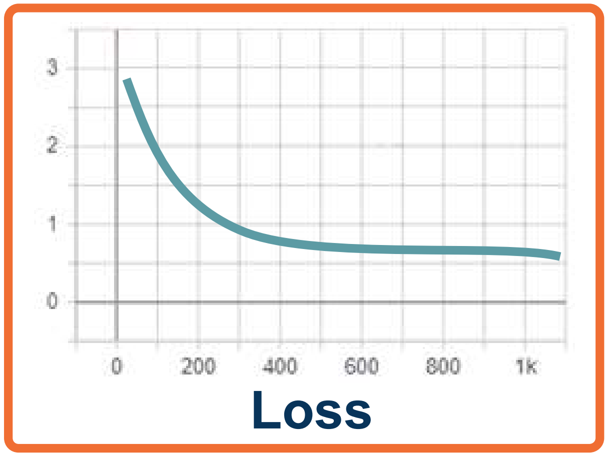
- **Example:** Cross entropy loss

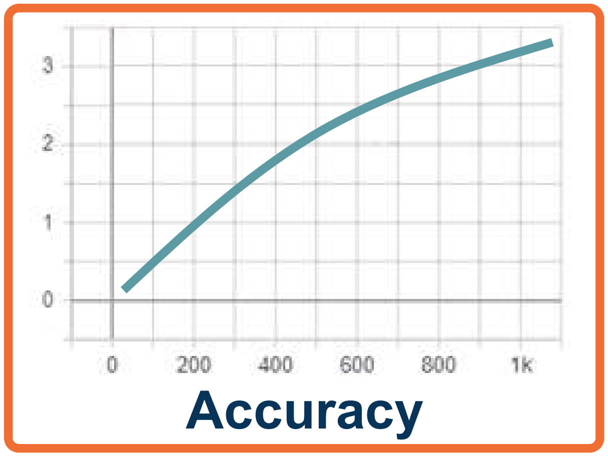
$$L = -log\, P(Y = y_i | X = x_i)$$

- **Accuracy** is measured based on:

$$argmax_i(P(Y = y_i\, | X = x_i))$$

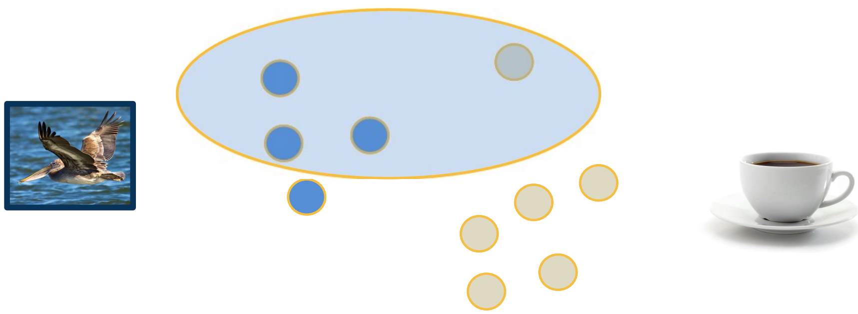
- Since the correct class score only has to be slightly higher, we can have **flat loss curves but increasing accuracy**!

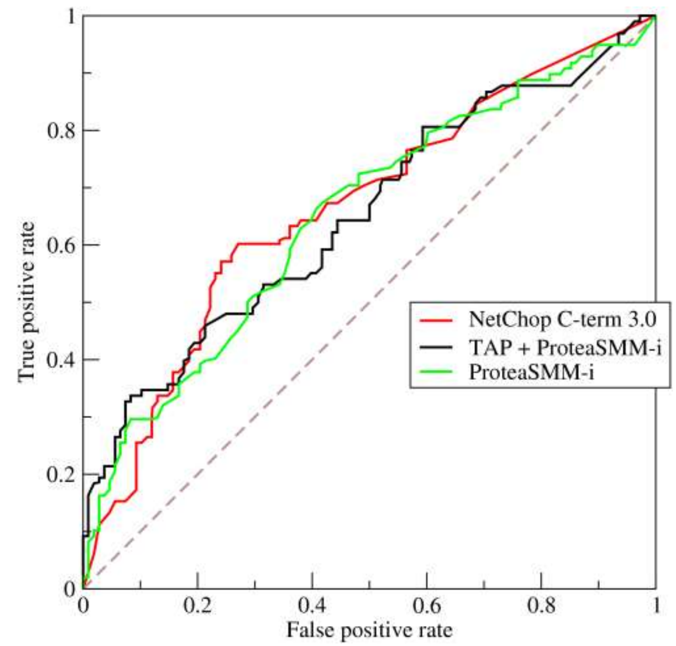




Example: Precision/Recall or ROC Curves

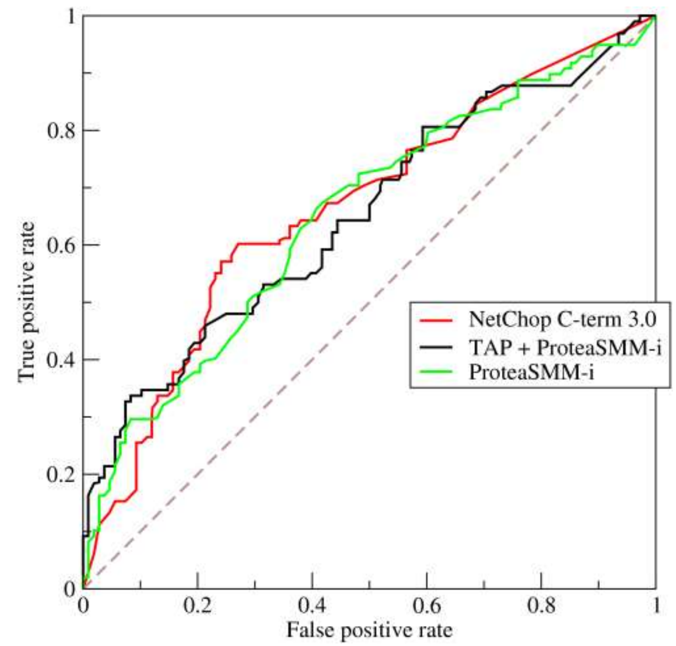
- **Precision/Recall curves** represent the inherent tradeoff between number of positive predictions and correctness of predictions
- **Definitions**
  - True Positive Rate: $\boldsymbol{TPR} = \frac{\boldsymbol{tp}}{\boldsymbol{tp+fn}}$
  - False Positive Rate: $\boldsymbol{FPR} = \frac{\boldsymbol{fp}}{\boldsymbol{fp+tn}}$
  - $\boldsymbol{Accuracy} = \frac{\boldsymbol{tp+tn}}{\boldsymbol{tp+tn+fp+fn}}$





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

- **Precision/Recall curves** represent the inherent tradeoff between number of positive predictions and correctness of predictions
- **Definitions**
  - True Positive Rate: $TPR = \frac{tp}{tp+fn}$
  - False Positive Rate: $FPR = \frac{fp}{fp+tn}$
  - $Accuracy = \frac{tp+tn}{tp+tn+fp+fn}$
- We can obtain a **curve** by varying the (probability) threshold**:**
  - **Area under the curve (AUC)** common single-number metric to summarize
- Mapping between this and loss is **not simple**!

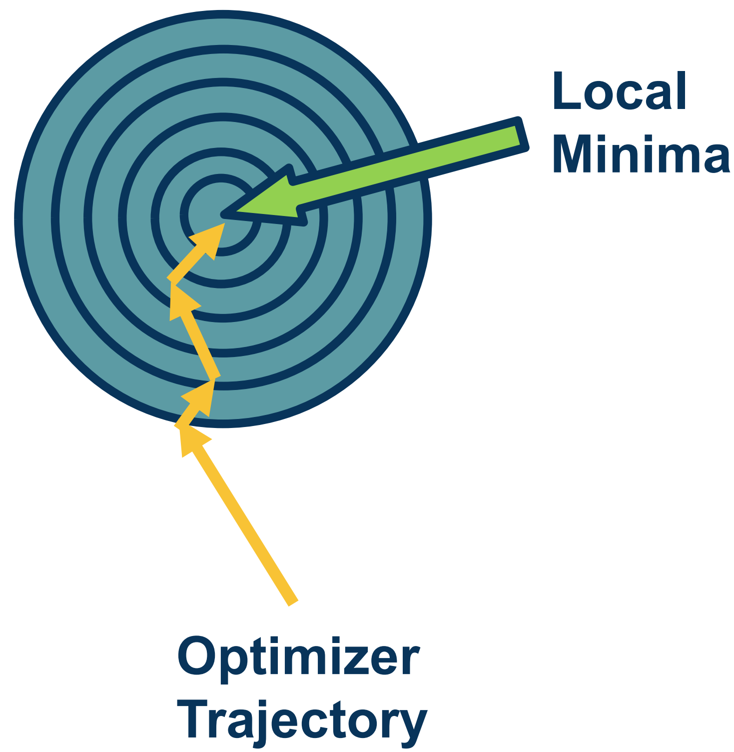


From https://en.wikipedia.org/wiki/Receiver_operating_characteristic

Example: Precision/Recall or ROC Curves

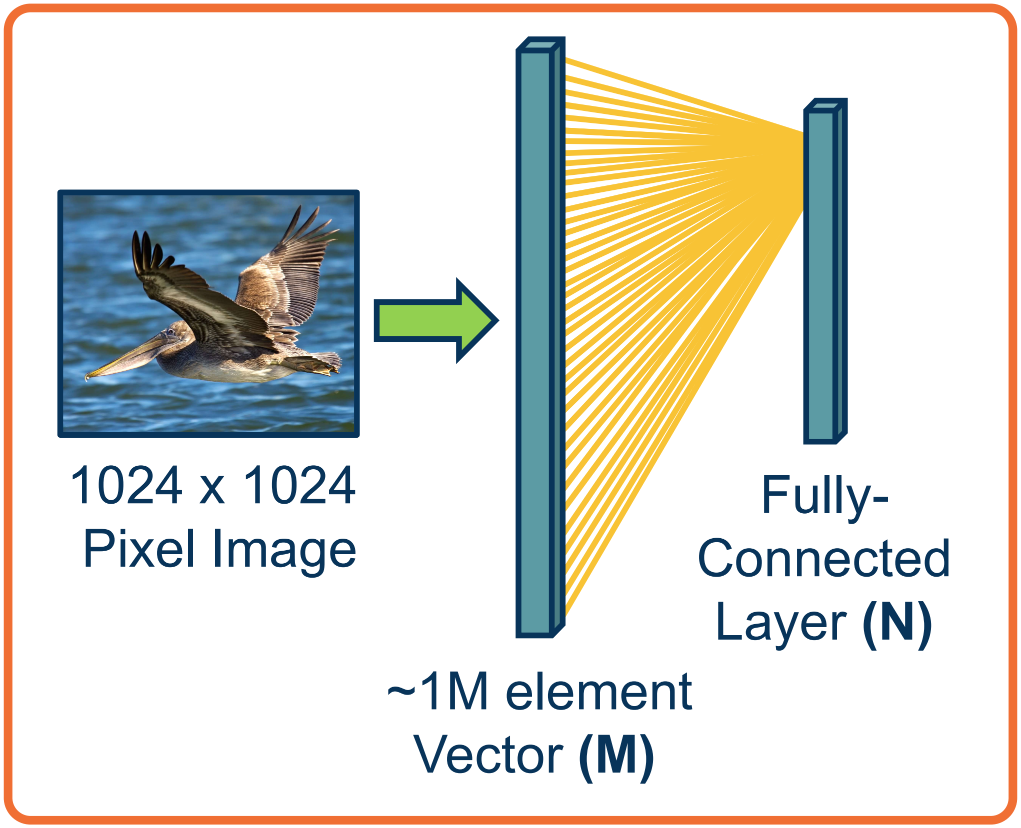
Resource:

- **A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay**, Leslie N. Smith



Convolution & Pooling

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

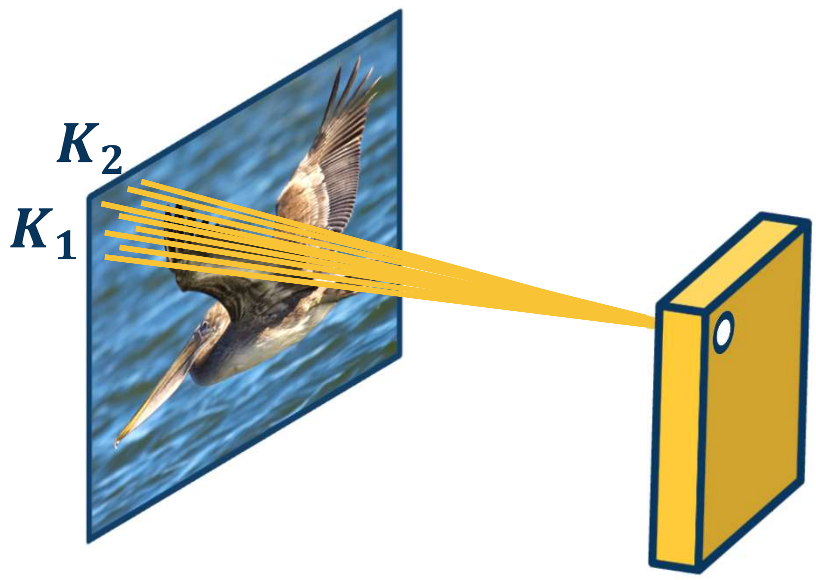
Is this necessary?

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?

Locality of Features



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

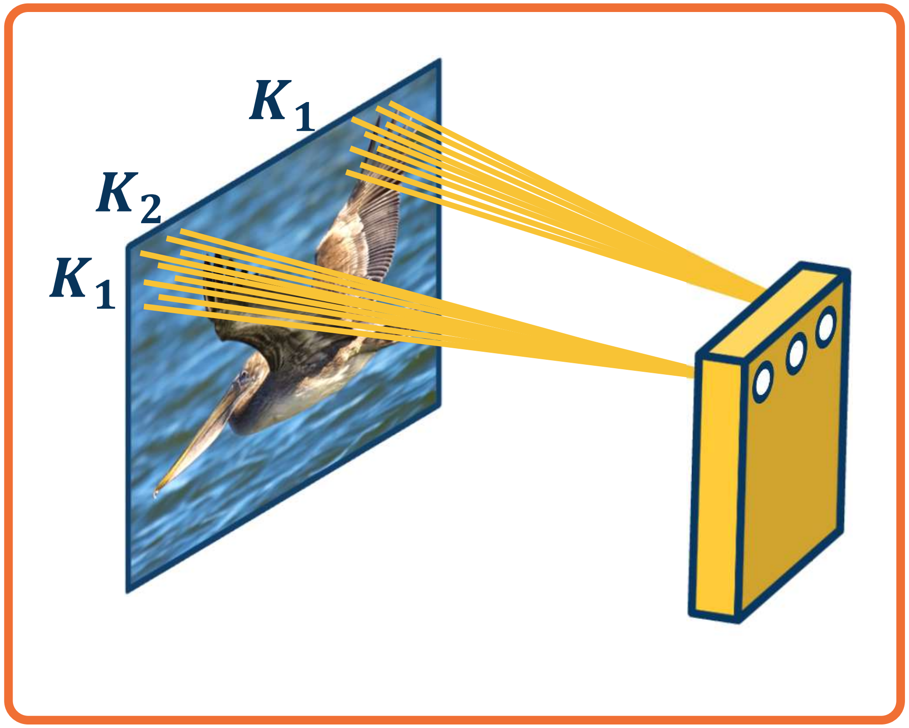
- Region from which a node receives input from is called its **receptive field**

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

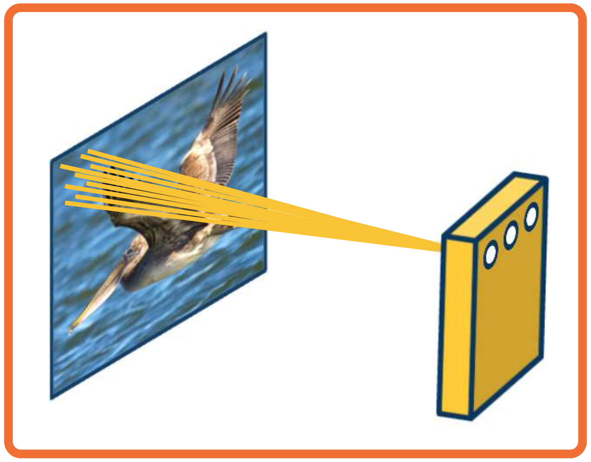
- No reason to think same feature (e.g. edge pattern) can't appear elsewhere
- Use same weights/parameters in computation graph **(shared weights)**

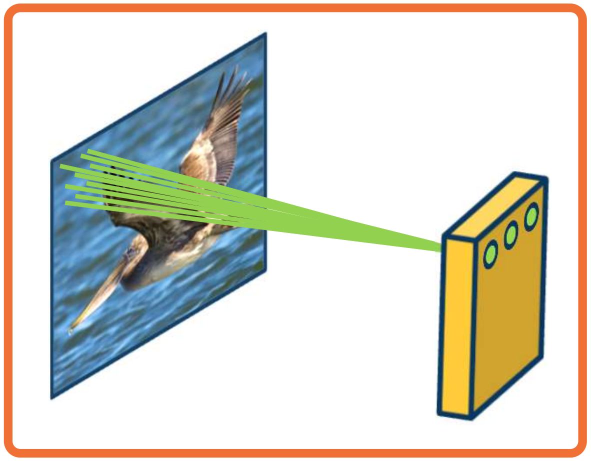
Advantages:

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

Idea 2: Shared Weights

Georgia Tech



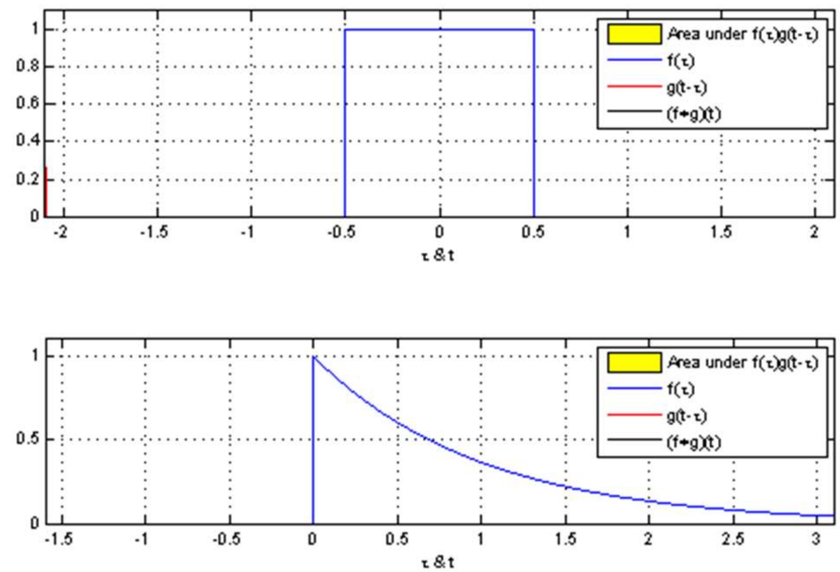


We can learn **many** such features for this one layer

- 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

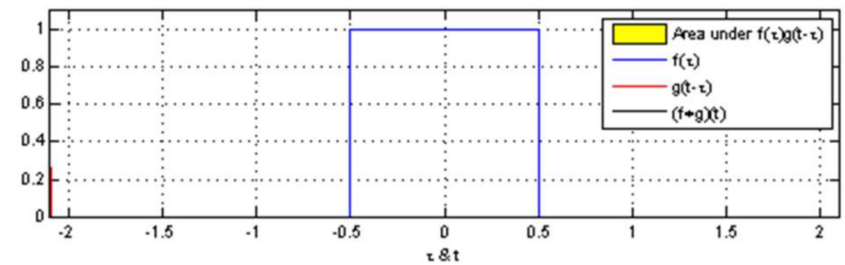
Idea 3: Learn Many Features

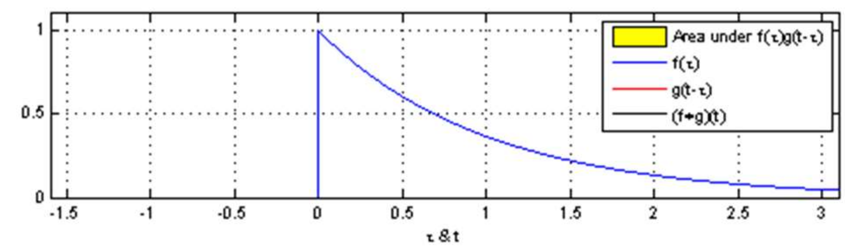
This operation is **extremely common** in electrical/computer engineering!



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

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





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

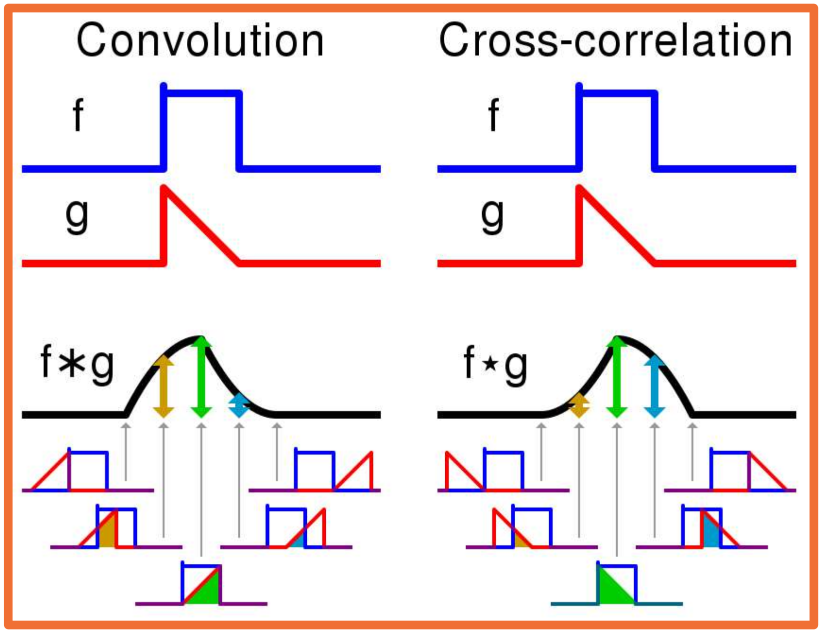
Convolution

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: $\boldsymbol{F \otimes (G \otimes I) = (F \otimes G) \otimes I}$

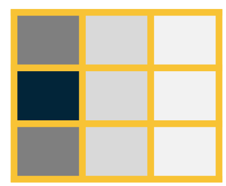
1D Convolution

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

$$\begin{aligned}
y_0 &= h_0 \cdot x_0 \\
y_1 &= h_1 \cdot x_0 + h_0 \cdot x_1 \\
y_2 &= h_2 \cdot x_0 + h_1 \cdot x_1 + h_0 \cdot x_2 \\
y_3 &= h_3 \cdot x_0 + h_2 \cdot x_1 + h_1 \cdot x_2 + h_0 \cdot x_3 \\
&\vdots
\end{aligned}$$

2D Convolution

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



2D Discrete Convolution

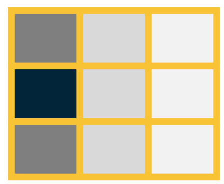
2D Discrete Convolution

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

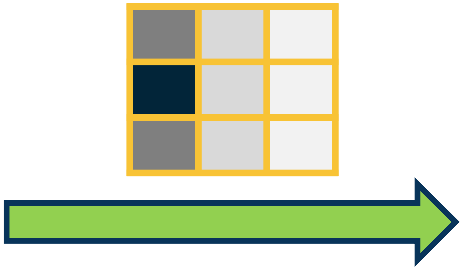
- Initialize kernel values randomly and optimize them!
- These are our parameters (plus a bias term per filter)

2D Convolution

Image

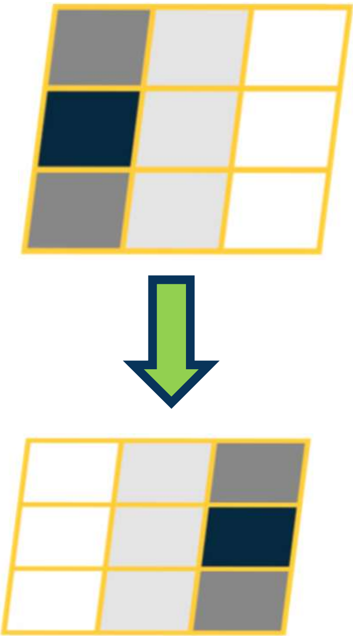
Kernel (or filter)

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

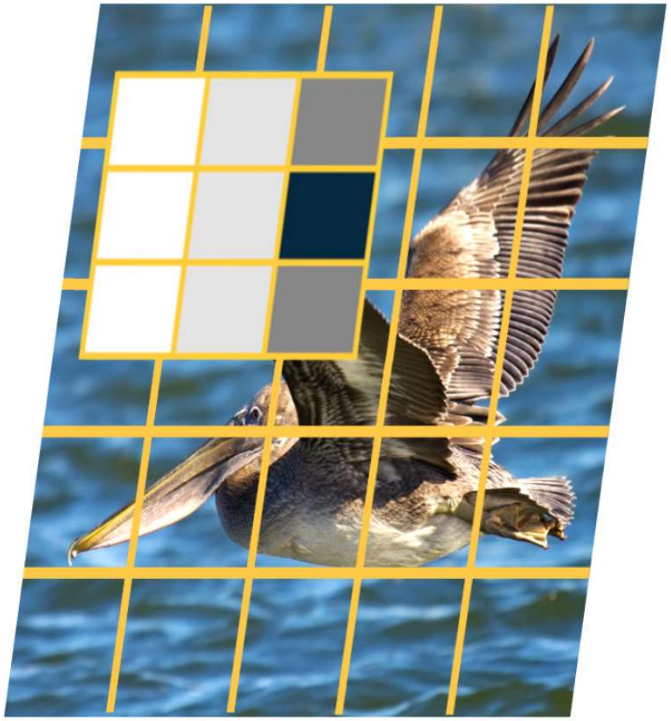


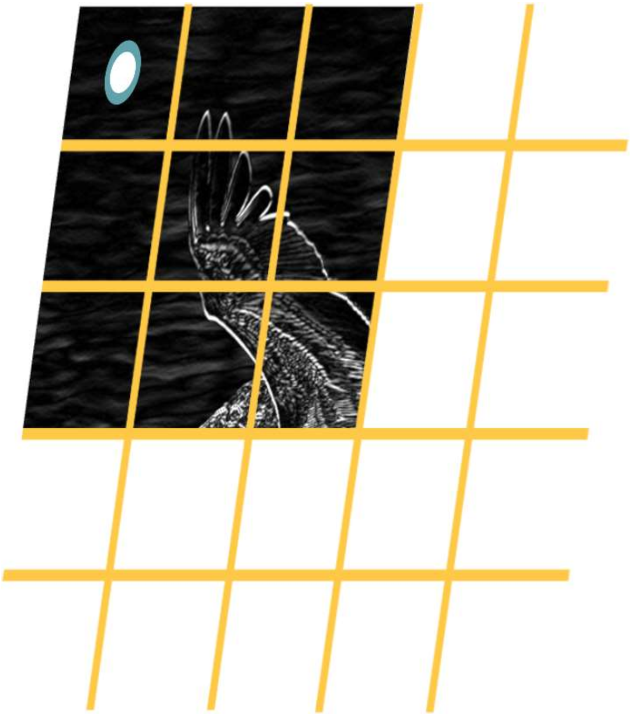
Output / filter / feature map

1. Flip kernel (rotate 180 degrees)



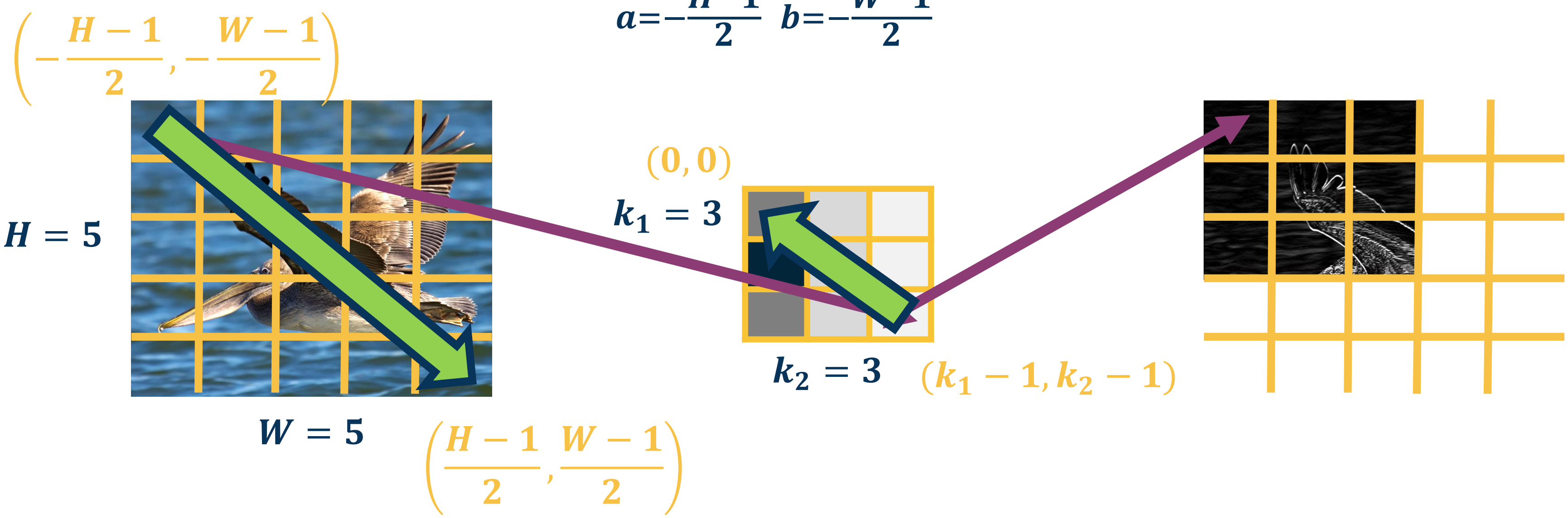
2. Stride along image





The Intuitive Explanation

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



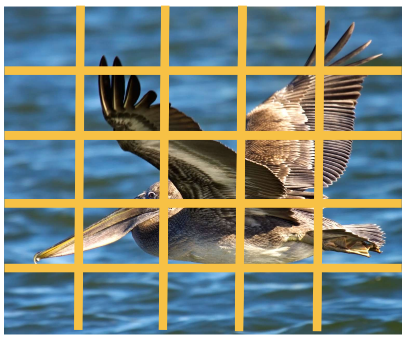
$$y(0,0) = x(-2,-2)k(2,2) + x(-2,-1)k(2,1) + x(-2,0)k(2,0) + x(-2,1)k(2,-1) + x(-2,2)k(2,-2) + \ldots$$

Mathematics of Discrete 2D Convolution

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

$(0,0)$

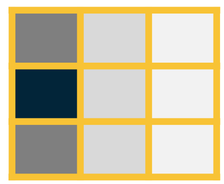
$H = 5$



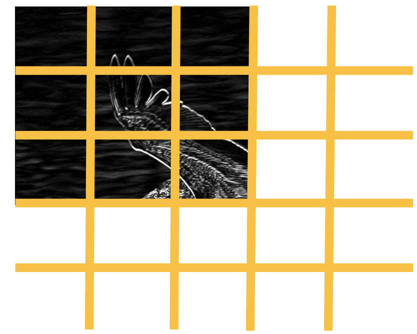
$W = 5$ $(H-1, W-1)$

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

$k_1 = 3$



$k_2 = 3$ $\left(\frac{k_1-1}{2}, \frac{k_2-1}{2}\right)$



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:

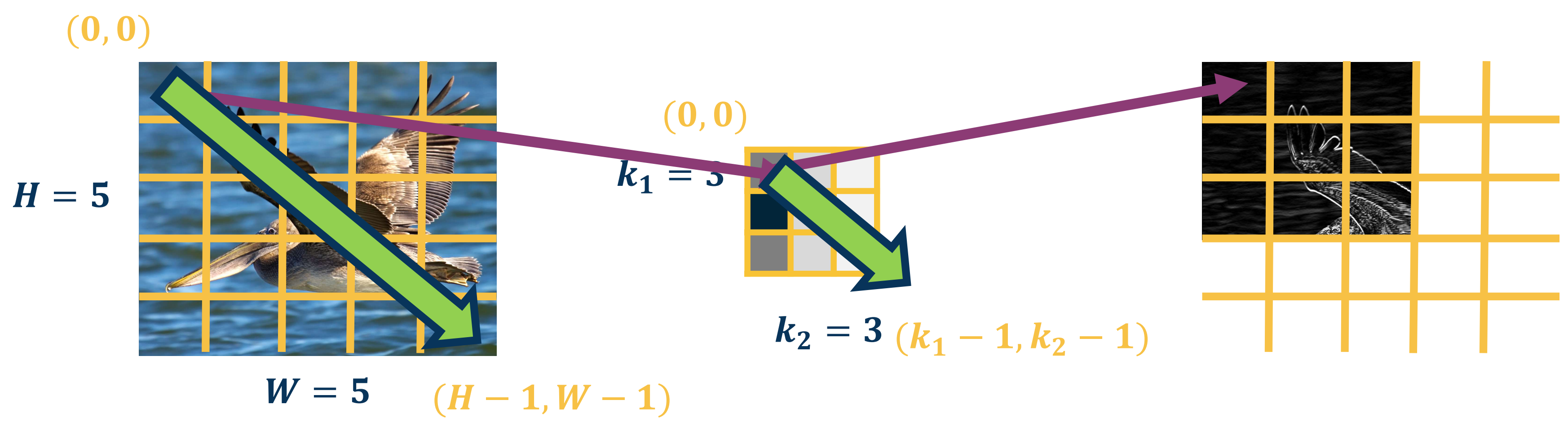
- Take the kernel, and rotate 180 degrees along center (sometimes referred to as “flip”)
- Perform cross-correlation
- (Just dot-product filter with image!)

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

$$\Downarrow$$

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

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

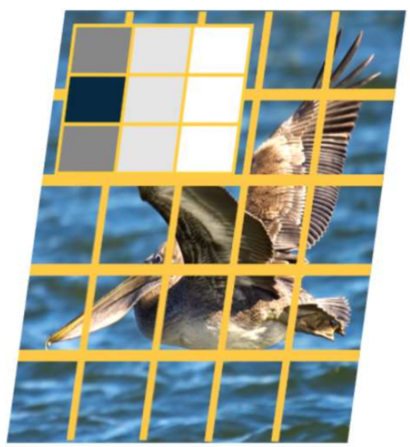


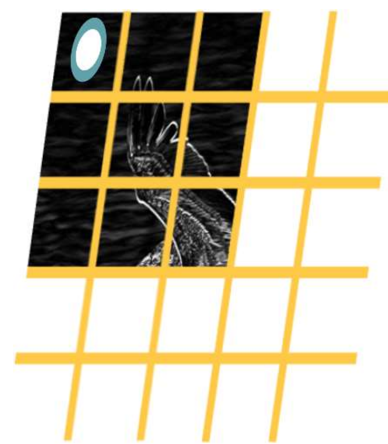
Since we will be learning these kernels, this change does not matter!

Cross-Correlation

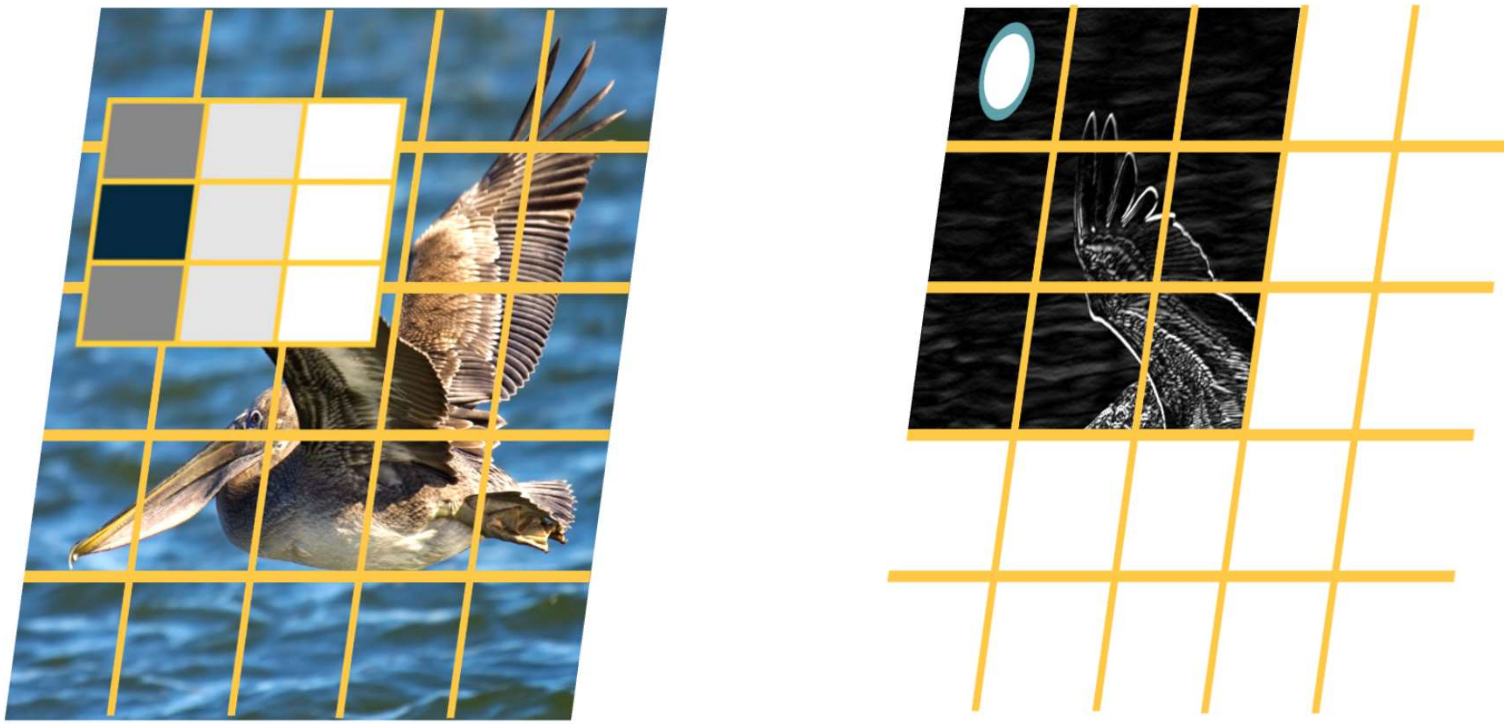
$$X(0:2,0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix} \qquad K' = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \Rightarrow X(0:2,0:2) \cdot K' = 65 + \text{bias}$$

Dot product
(element-wise multiply and sum)

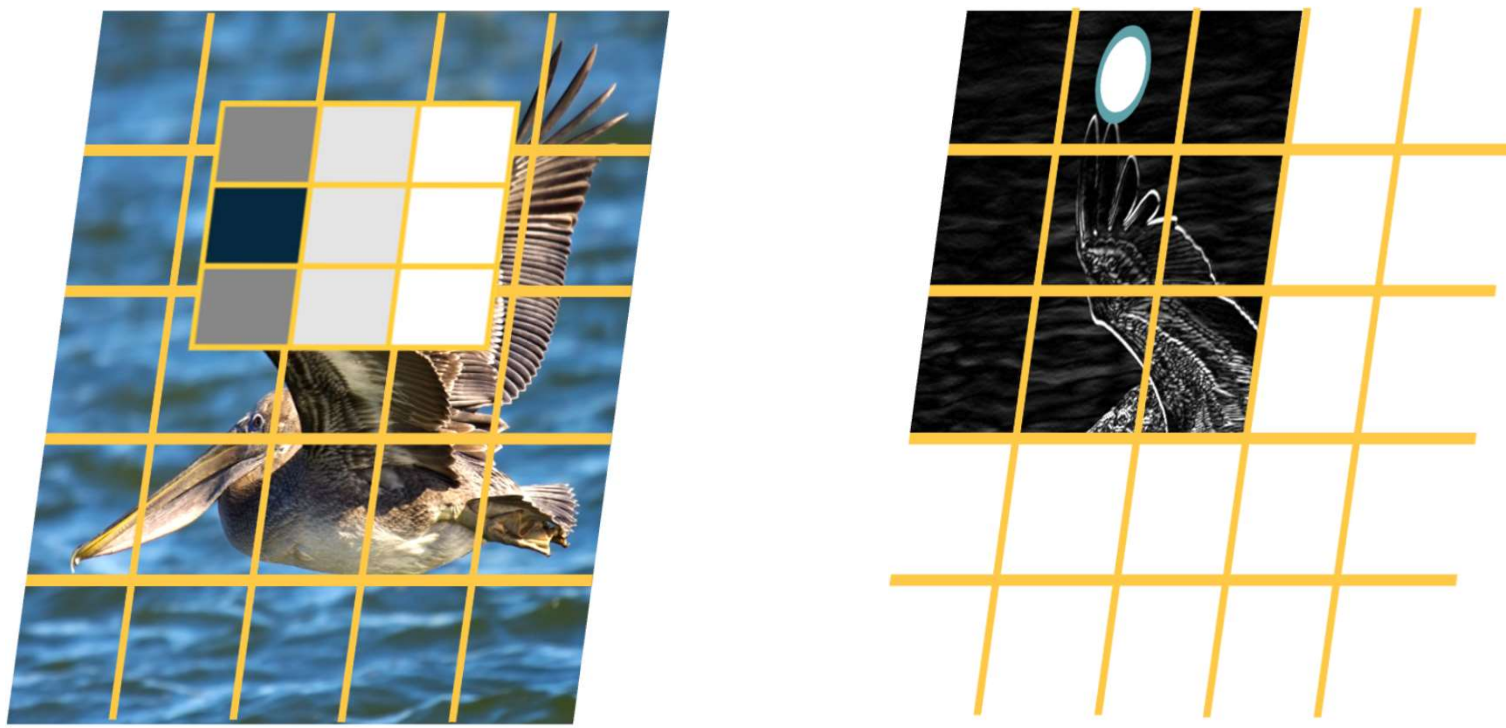




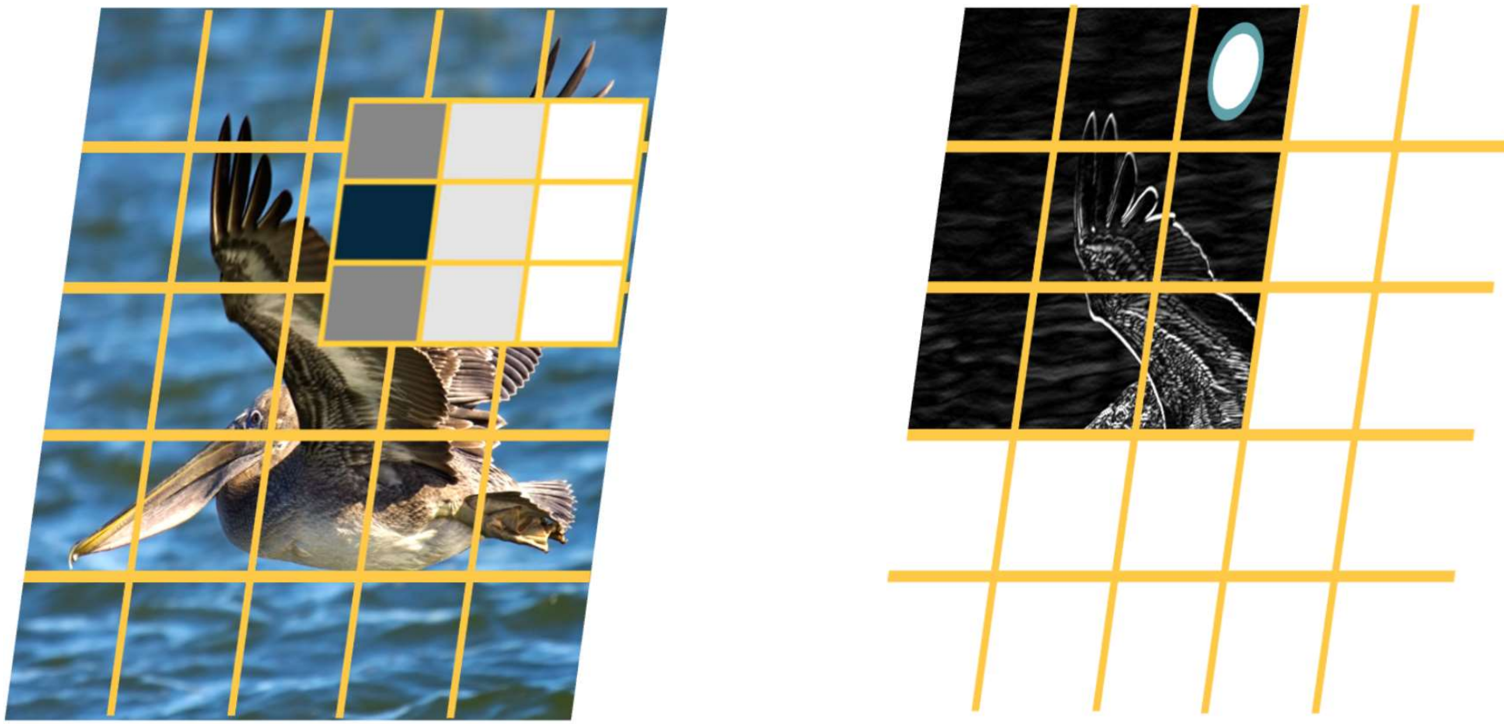
Cross-Correlation



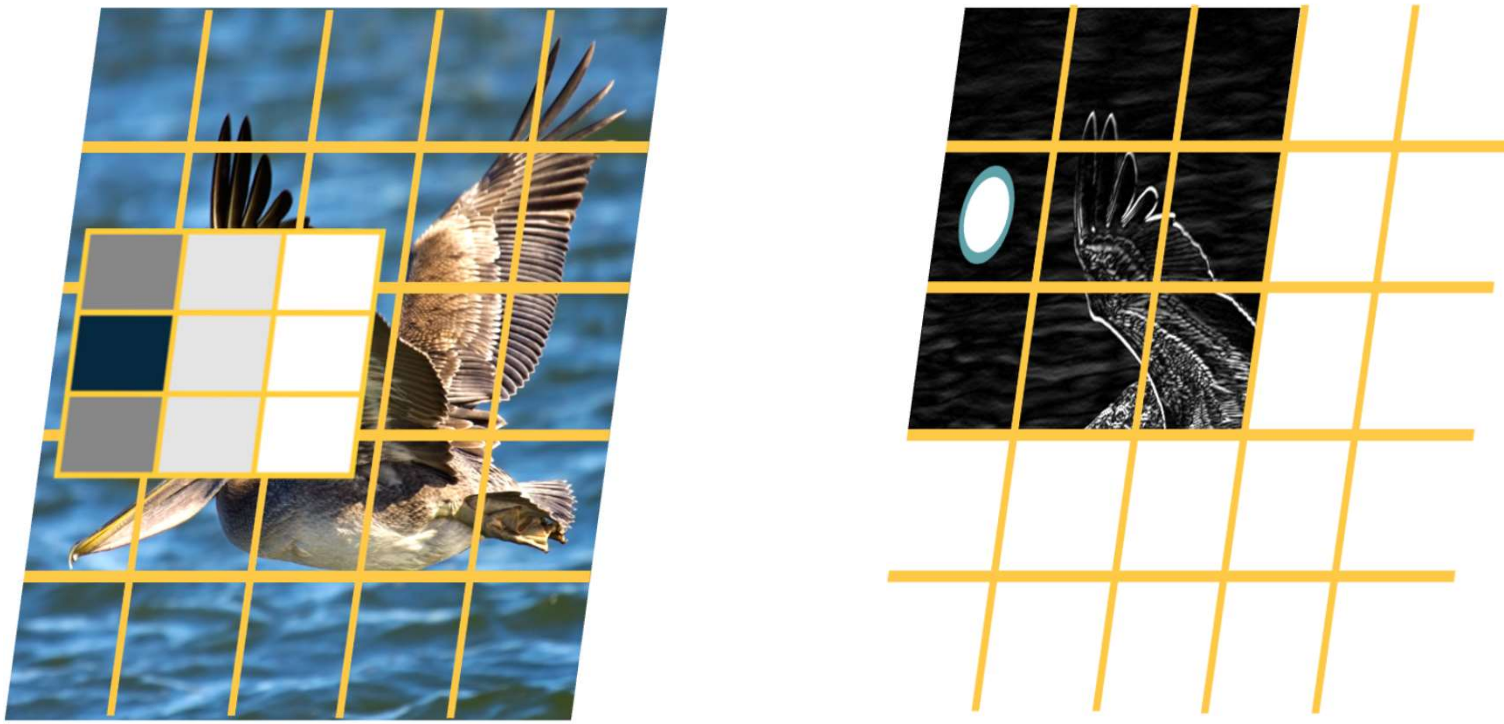
Convolution and Cross-Correlation



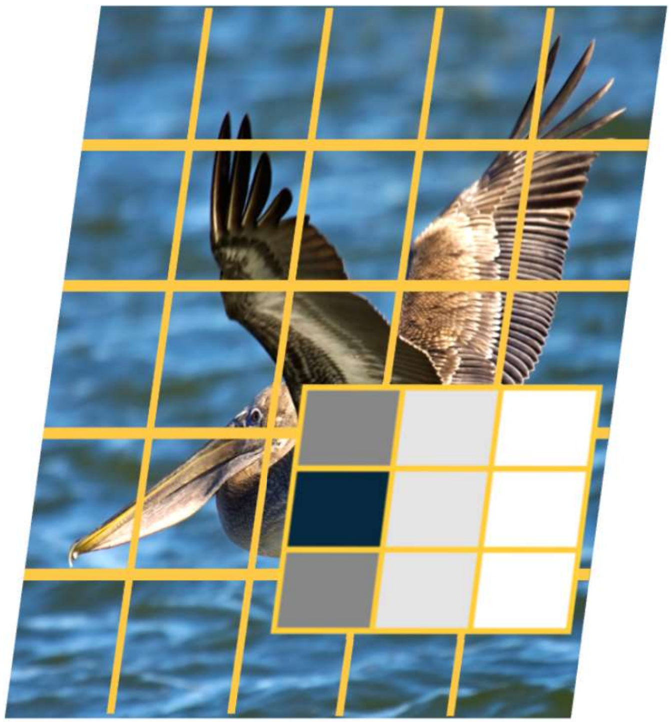
Convolution and Cross-Correlation

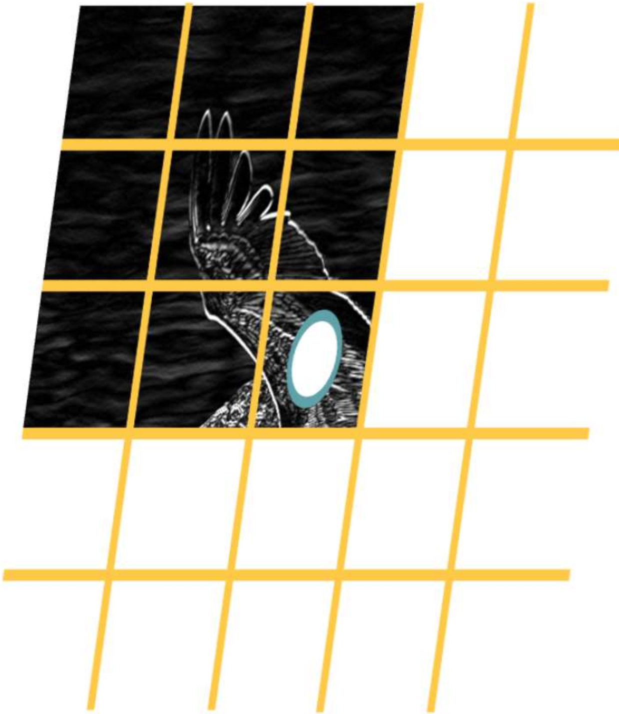


Convolution and Cross-Correlation



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

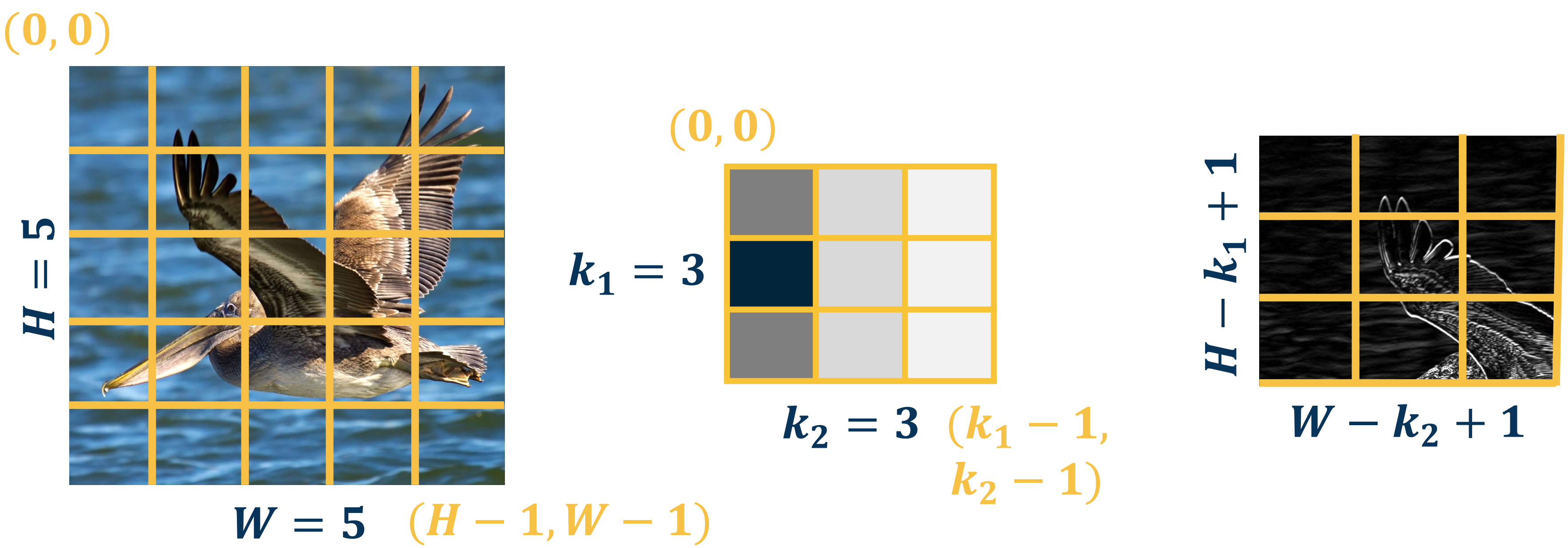
- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int* or *tuple*) – Size of the convolving kernel
- **stride** (*int* or *tuple*, *optional*) – Stride of the convolution. Default: 1
- **padding**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

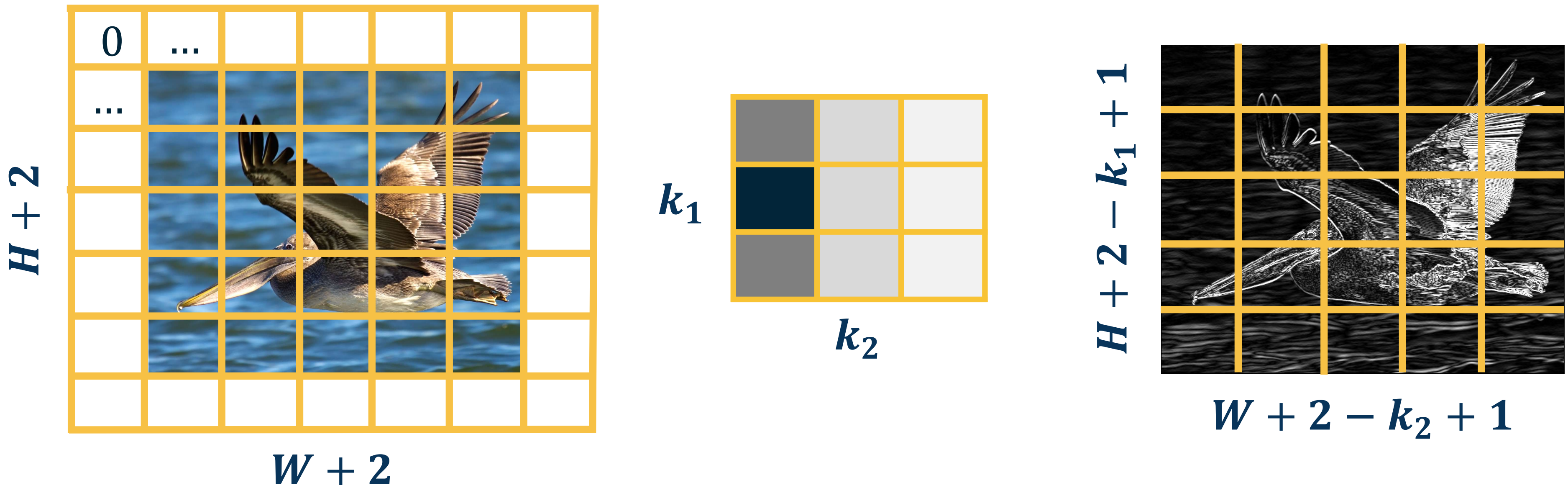


Valid Convolution

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

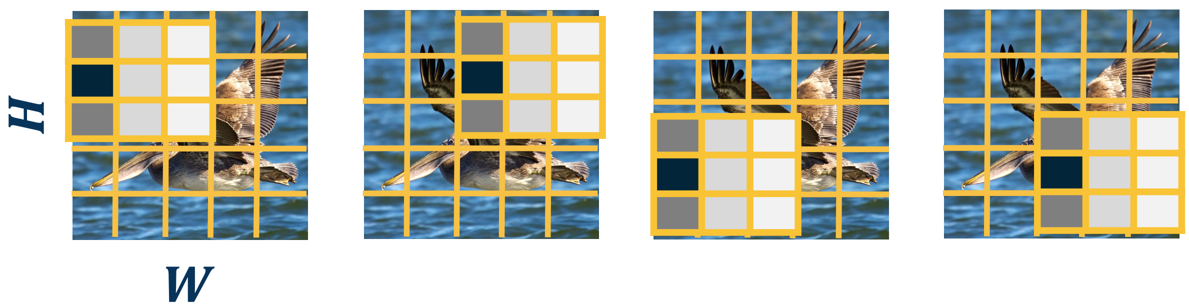


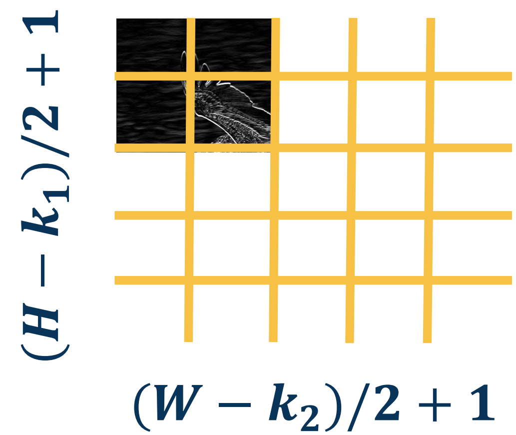
Stride

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

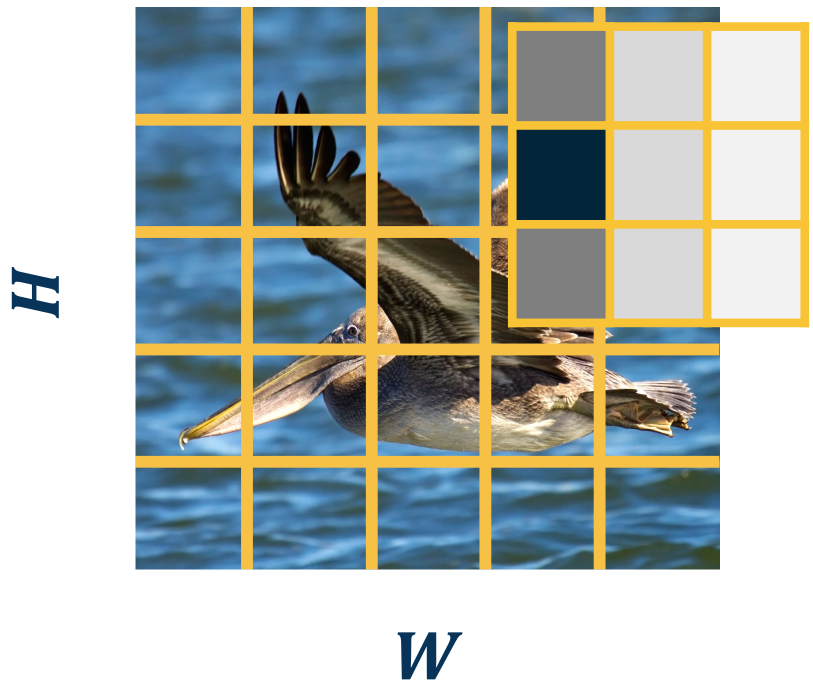
- This can potentially result in **loss of information**
- Can be used for **dimensionality reduction** (not recommended)

Stride = 2 (every other pixel)





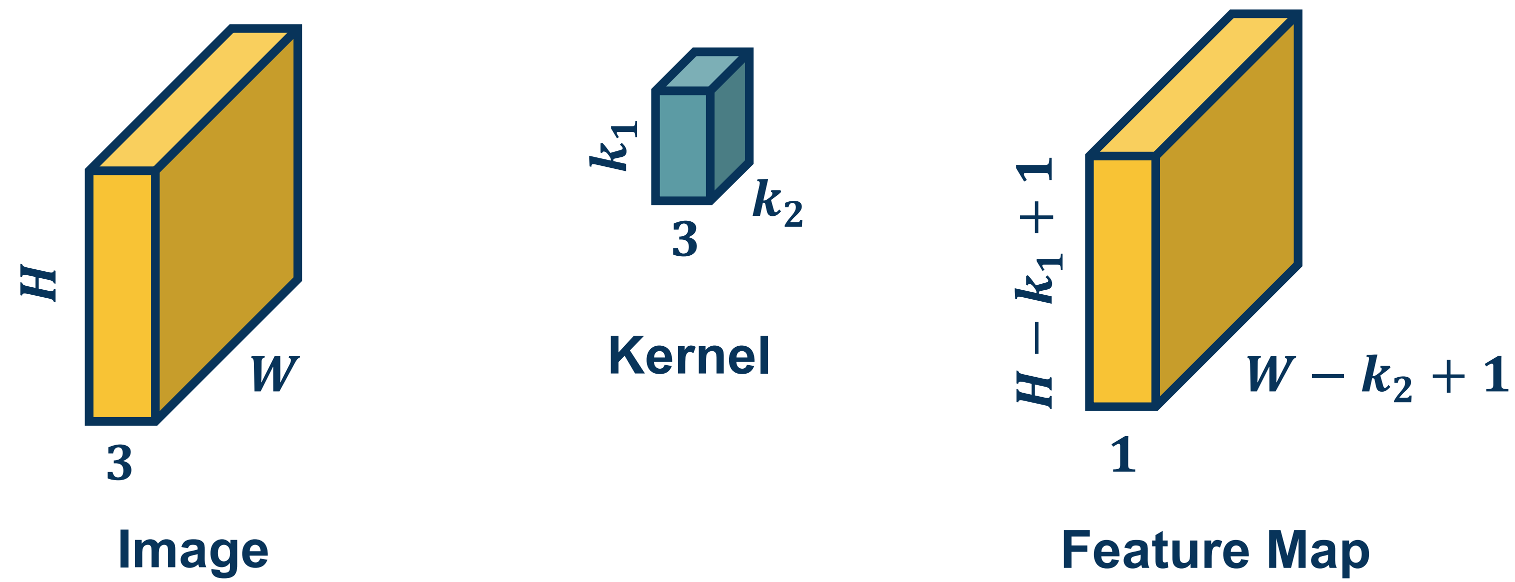
Stride can result in **skipped pixels**, e.g. stride of 3 for 5x5 input



Invalid Stride

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

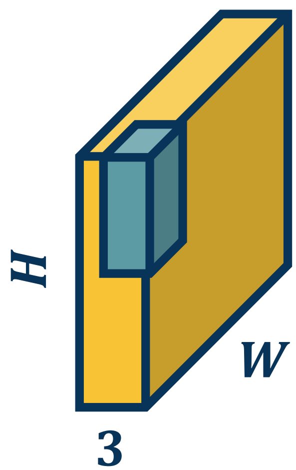
- In such cases, we have **3-channel kernels**!



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)**

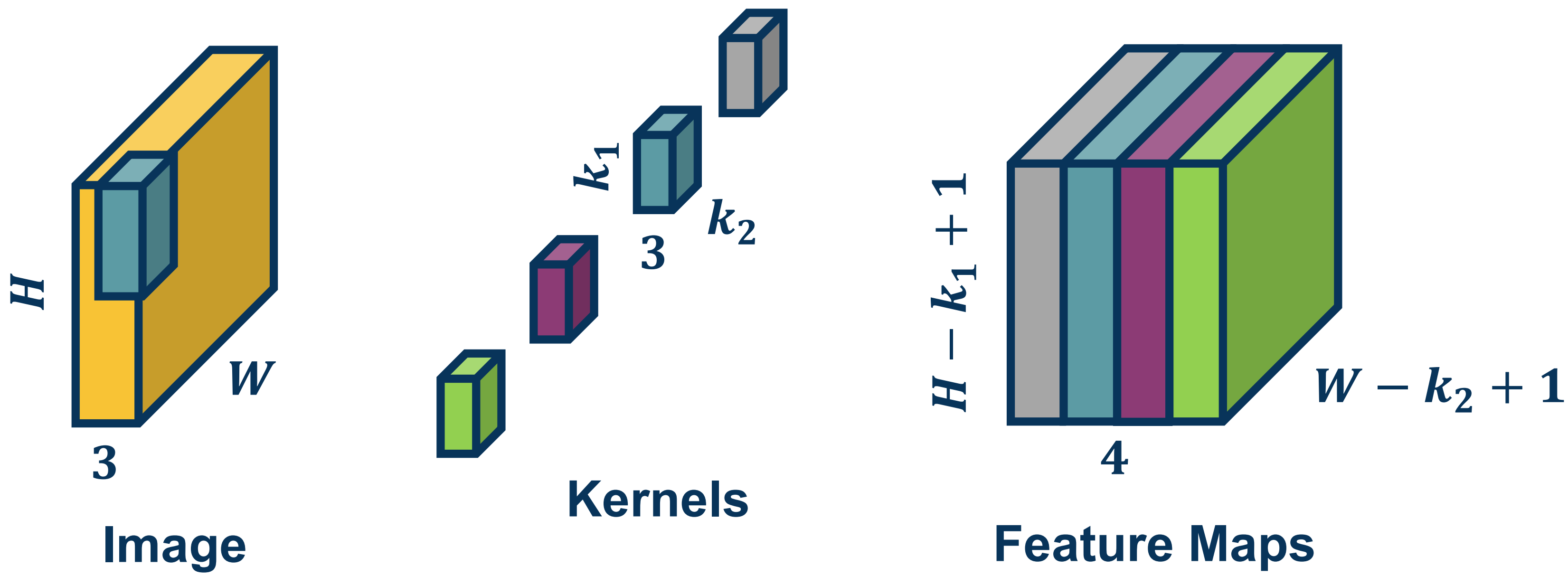
- Except with $\boldsymbol{k_1 * k_2 * 3}$ values

Operation of Multi-Channel Input

We can have **multiple kernels per layer**

- We stack the feature maps together at the output

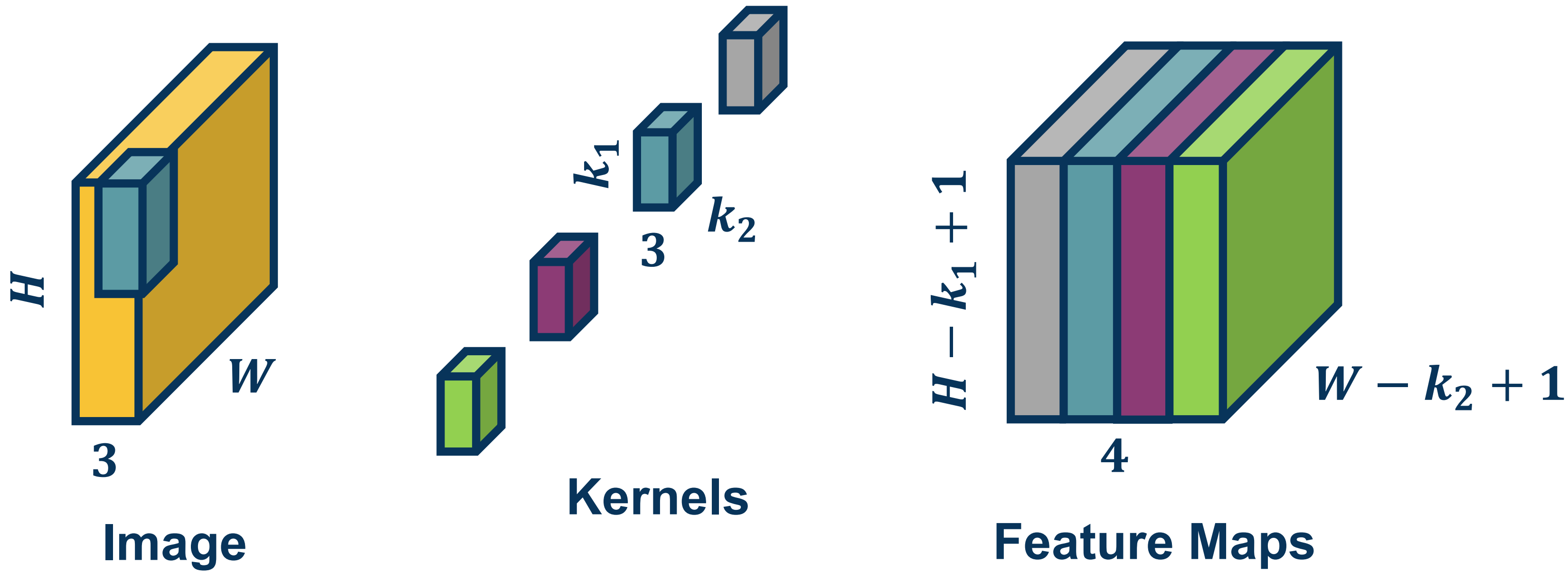
Number of channels in output is equal to *number of kernels*



Multiple Kernels

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

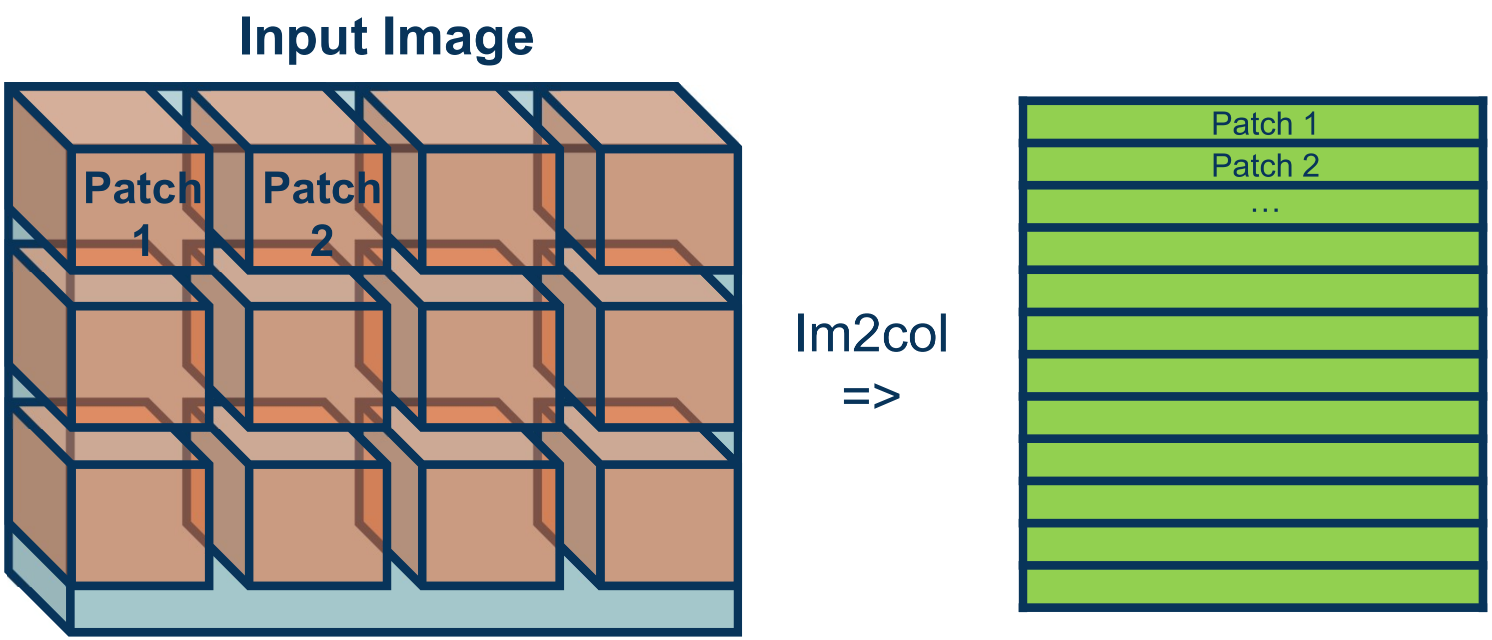
- Example:
  $k_1 = 3, k_2 = 3,\ N = 4\ input\ channels = 3$, then $(3 * 3 * 3 + 1) * 4 =$112



Number of Parameters

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

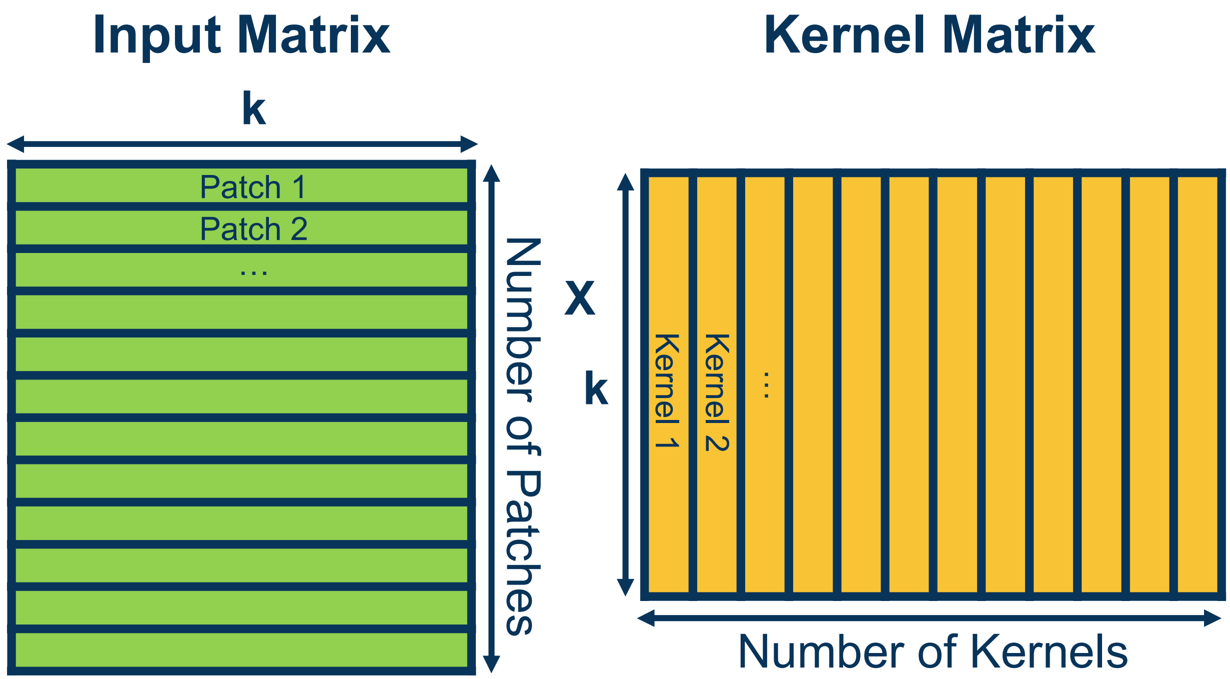
- **Step 1:** Lay out image patches in vector form (note can overlap!)



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/