Topics:

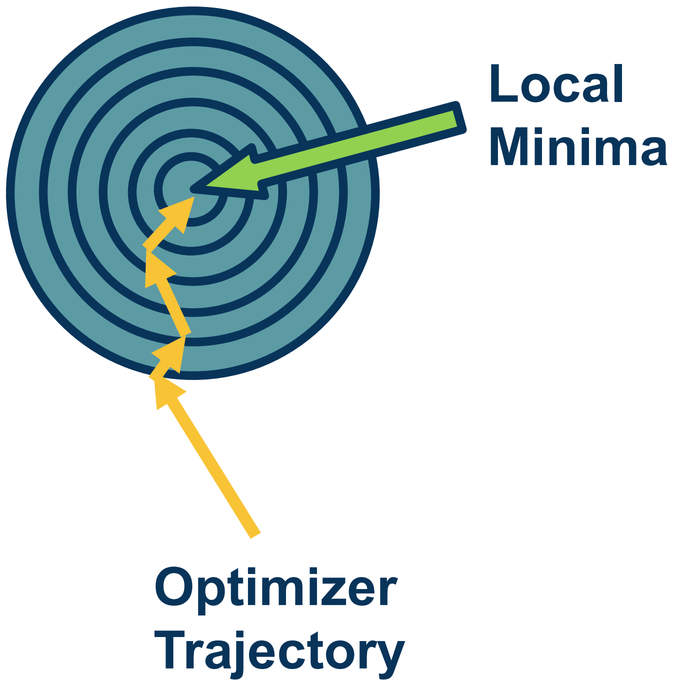
- Optimization (Cont)
- Convolution

CS 4644-DL / 7643-A

ZSOLT KIRA

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

Optimizers

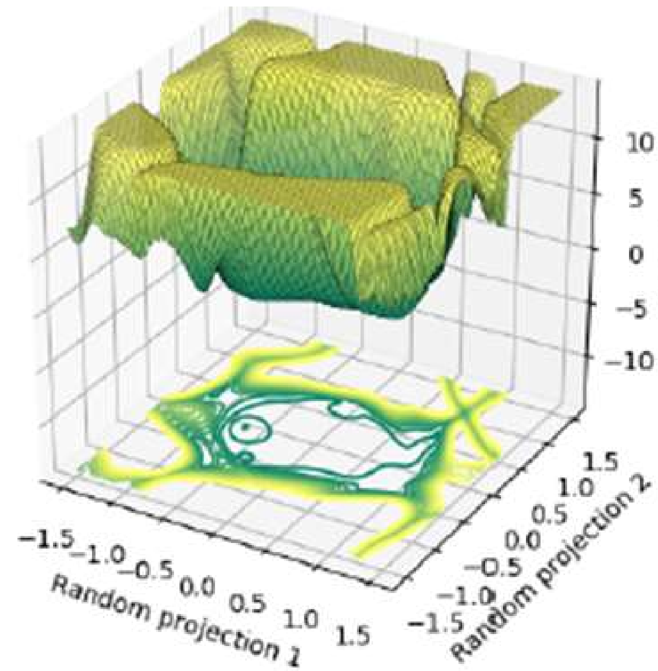
Georgia Tech

Deep learning involves **complex, compositional, non-linear functions**

The **loss landscape** is extremely **non-convex** as a result

There is **little direct theory** and a **lot of intuition/rules of thumbs** instead

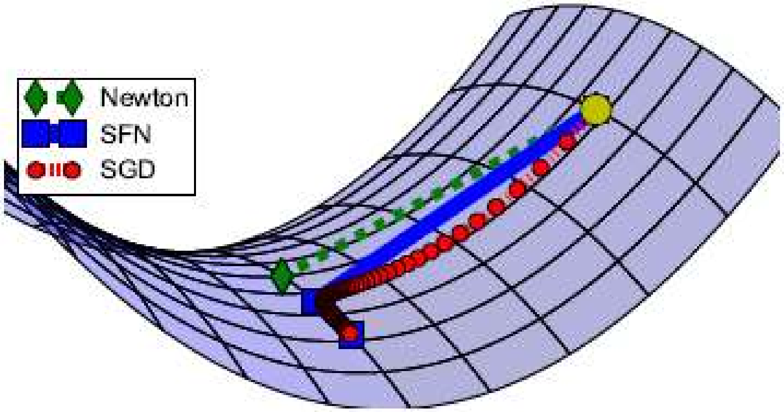
- Some insight can be gained via theory for simpler cases (e.g. convex settings)



It used to be thought that **existence of local minima is the main issue** in optimization

There are other **more impactful issues**:

- Noisy gradient estimates
- Saddle points
- Ill-conditioned loss surface



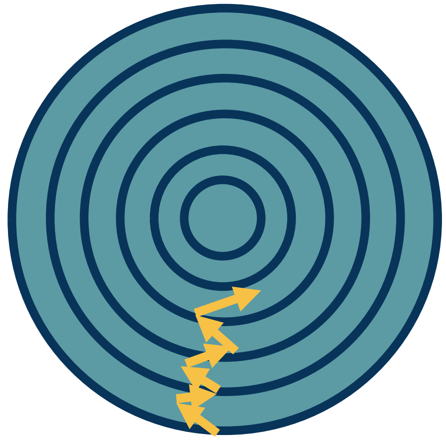
Saddle Point

From: Identifying and attacking the saddle point problem in high-dimensional non-convex optimization, Dauphi et al., 2014.

Noisy Gradients

- We use a **subset of the data at each iteration** to calculate the loss (& gradients)
- This is an **unbiased** estimator but can have high variance
- This results in **noisy steps** in gradient descent

$$L = \frac{1}{M} \sum L\left(f(x_i, W), y_i\right)$$

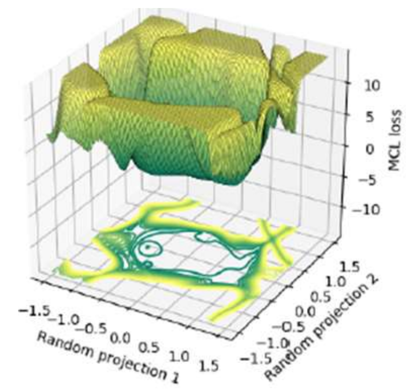


Several **loss surface geometries** are difficult for optimization

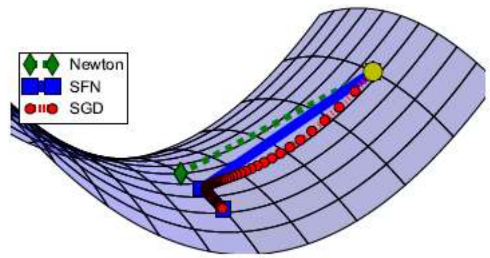
Several types of minima: Local minima, plateaus, saddle points

Saddle points are those where the gradient of orthogonal directions are zero

- But they **disagree** (it's min for one, max for another)



Plateau

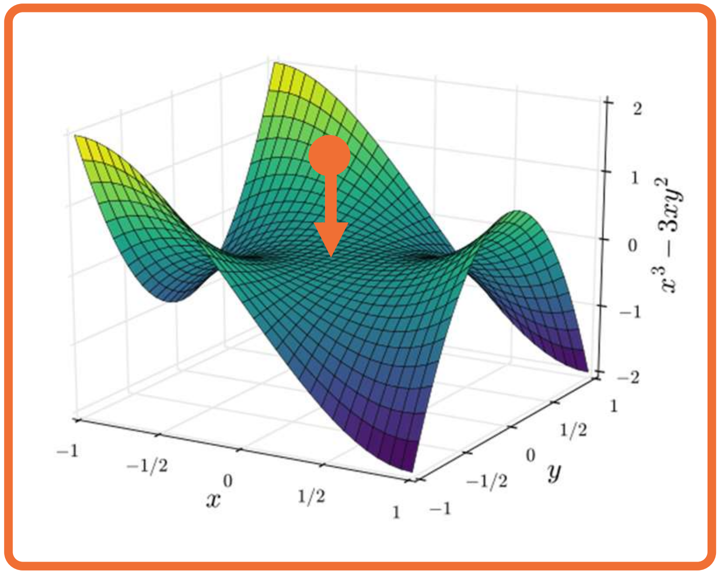


Saddle Point

- Gradient descent takes a step in the **steepest direction** (negative gradient)

$$w_i = w_{i-1} - \alpha \frac{\partial L}{\partial w_i}$$

- **Intuitive idea:** Imagine a ball rolling down loss surface, and use **momentum** to pass flat surfaces



$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial w_{i-1}}$$ **Update Velocity (starts as 0, $\beta = 0.99$)**

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

- Generalizes SGD ($\beta = 0$)

Adding Momentum

- Velocity term is an **exponential moving average** of the gradient

$$v_i = \beta v_{i-1} + \frac{\partial L}{\partial w_{i-1}}$$

$$v_i = \beta(\beta\, v_{i-2} + \frac{\partial L}{\partial w_{i-2}}) + \frac{\partial L}{\partial w_{i-1}}$$

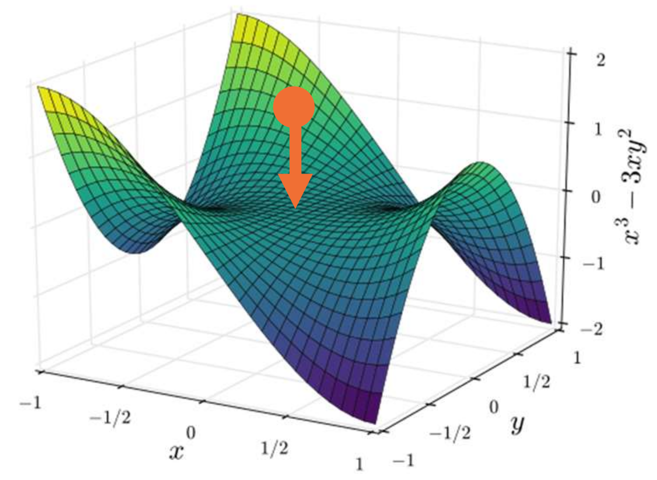
$$= \beta^2 v_{i-2} + \beta \frac{\partial L}{\partial w_{i-2}} + \frac{\partial L}{\partial w_{i-1}}$$

- There is a **general class of accelerated gradient methods**, with some theoretical analysis (under assumptions)

Equivalent formulation:

$$v_i = \beta v_{i-1} - \alpha \frac{\partial L}{\partial w_{i-1}}$$ **Update Velocity (starts as 0)**

$$w_i = w_{i-1} + v_i$$ **Update Weights**



Equivalent Momentum Update

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**

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

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

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

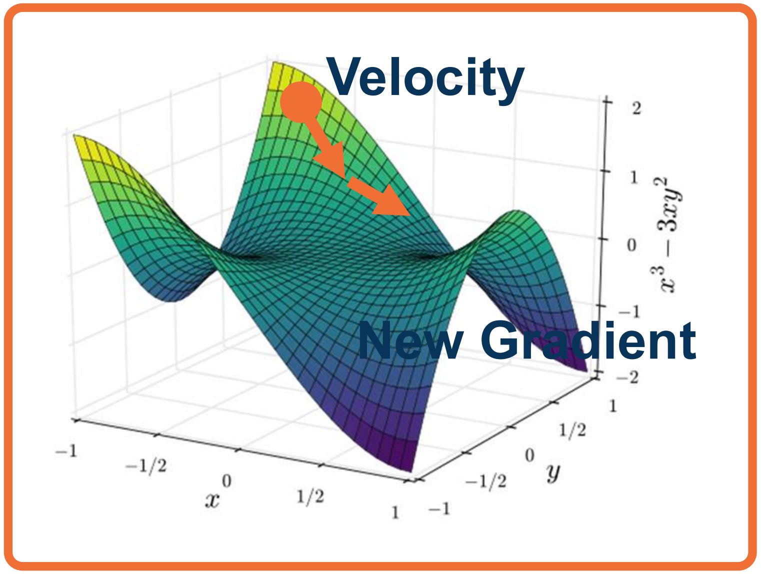


Figure Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Momentum

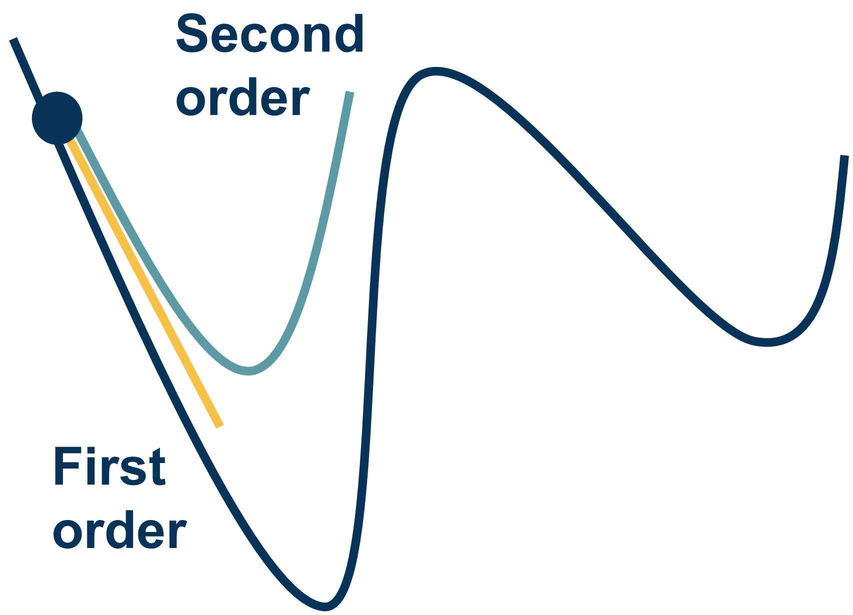
Note there are **several equivalent formulations** across deep learning frameworks!

Resource:
https://medium.com/the-artificial-impostor/sgd-implementation-in-pytorch-4115bcb9f02c

- Various mathematical ways to **characterize the loss landscape**
- If you liked **Jacobians**… meet the

$$\mathbf{H} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Gives us information about the **curvature of the loss surface**

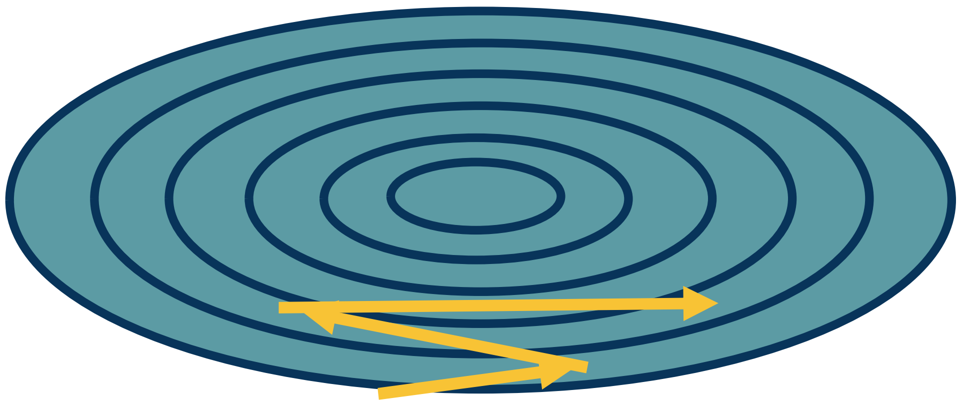


Condition number is the ratio of the largest and smallest eigenvalue

- Tells us how different the curvature is along different dimensions

If this is high, SGD will make **big** steps in some dimensions and **small** steps in other dimension

Second-order optimization methods divide steps by curvature, but expensive to compute



Condition Number

Per-Parameter Learning Rate

Idea: Have a dynamic learning rate for each weight

Several flavors of **optimization algorithms**:

- RMSProp
- Adagrad
- Adam
- …

SGD can achieve similar results in many cases but with much more tuning

Idea: Use gradient statistics to reduce learning rate across iterations

Denominator: Sum up gradients over iterations

Directions with **high curvature will have higher gradients**, and learning rate will reduce

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

$$w_i = w_{i-1} - \frac{\alpha}{\sqrt{G_i} + \epsilon} \frac{\partial L}{\partial w_{i-1}}$$

As gradients are accumulated learning rate will go to zero

Duchi, et al., "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization"

Adagrad

Solution: Keep a moving average of squared gradients!

Does not saturate the learning rate

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

$$w_i = w_{i-1} - \frac{\alpha}{\sqrt{G_i + \epsilon}} \frac{\partial L}{\partial w_{i-1}}$$

RMSProp

Combines ideas from above algorithms

Maintains both first and second moment statistics for gradients

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

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

$$w_i = w_{i-1} - \frac{\alpha \, v_i}{\sqrt{G_i + \epsilon}}$$

But unstable in the beginning (one or both of moments will be tiny values)

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Solution: Time-varying bias correction

Typically $\boldsymbol{\beta_1 = 0.9,\ \ \beta_2 = 0.999}$

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

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

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

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

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

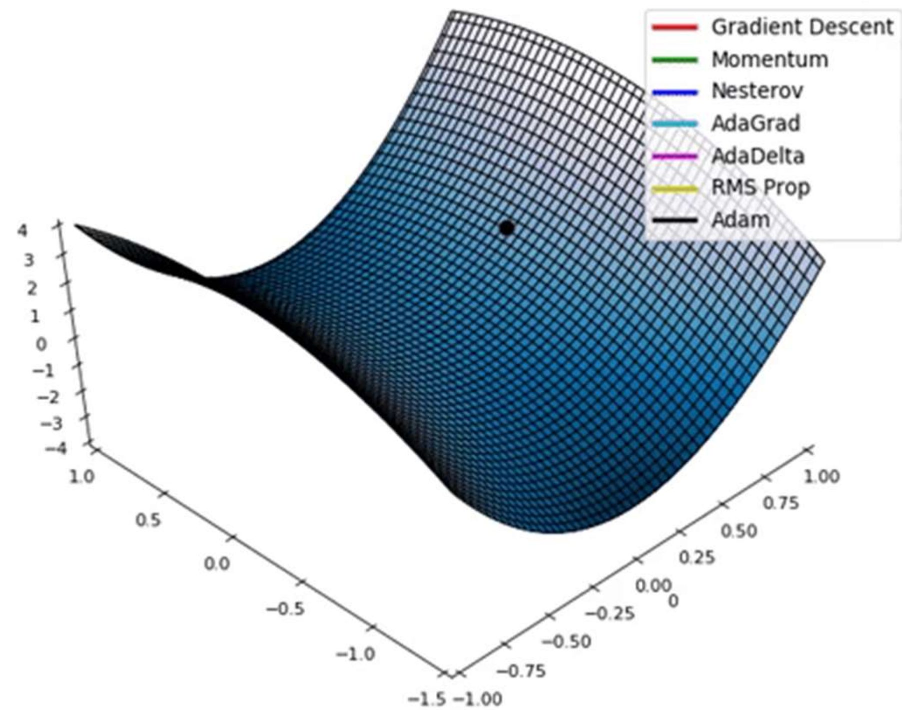
Adam

Optimizers behave differently **depending on landscape**

Different behaviors such as **overshooting, stagnating, etc.**

Plain SGD+Momentum can generalize better than adaptive methods, but requires more tuning

- **See:** *Luo et al., Adaptive Gradient Methods with Dynamic Bound of Learning Rate, ICLR 2019*



From: https://mlfromscratch.com/optimizers-explained/#/

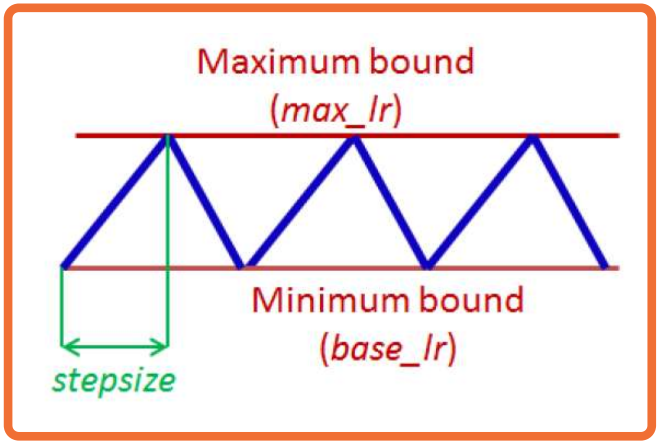
Behavior of Optimizers

First order optimization methods have **learning rates**

Theoretical results rely on **annealed learning rate**

Several schedules that are typical:

- Graduate student!
- Step scheduler
- Exponential scheduler
- Cosine scheduler



From: Leslie Smith, "Cyclical Learning Rates for Training Neural Networks"

Learning Rate Schedules

Regularization

Many **standard regularization methods** still apply!

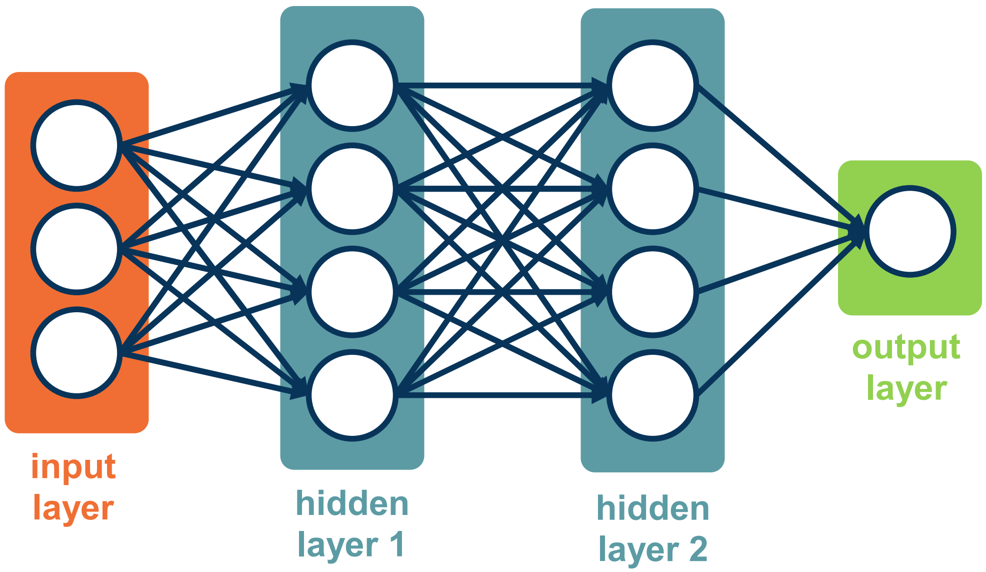
L1 Regularization

$$L = |y - Wx_i|^2 + \lambda|W|$$

where $|W|$ is element-wise

Example regularizations:

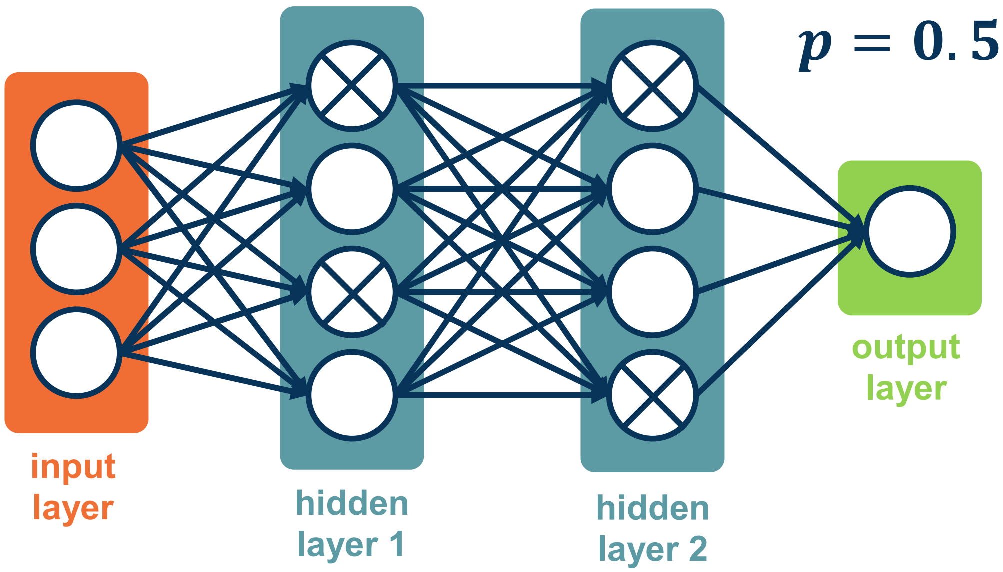
- L1/L2 on weights (encourage small values)
- L2: $L = |y - Wx_i|^2 + \lambda|W|^2$ (weight decay)
- Elastic L1/L2: $|y - Wx_i|^2 + \alpha|W|^2 + \beta|W|$



Problem: Network can learn to rely strong on a few features that work really well

- May cause **overfitting** if not representative of test data

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



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

- Activations of deactivated nodes are essentially zero

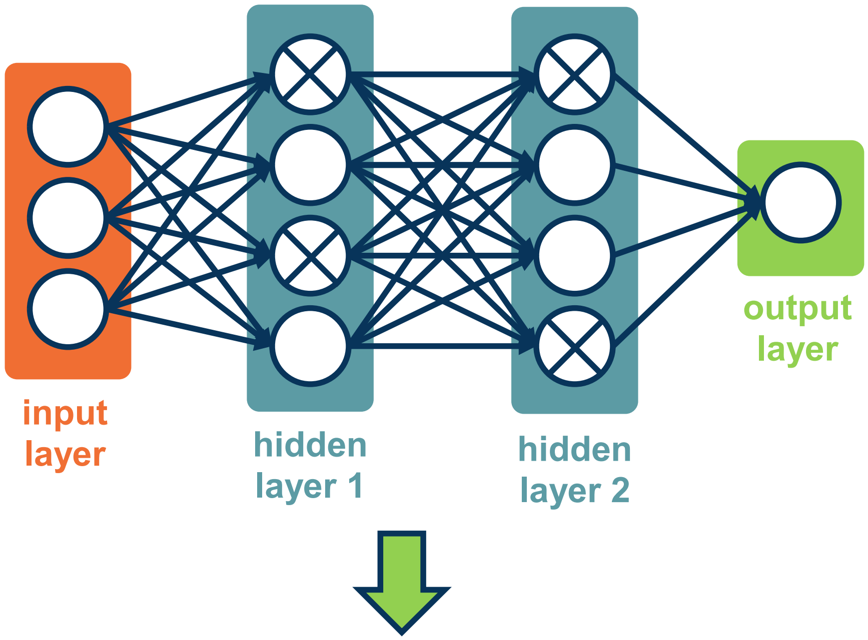
Choose whether to mask out a particular node **each iteration**

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

Dropout Regularization

- In practice, implement with a **mask** calculated each iteration

- During testing, no nodes are dropped



$$\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

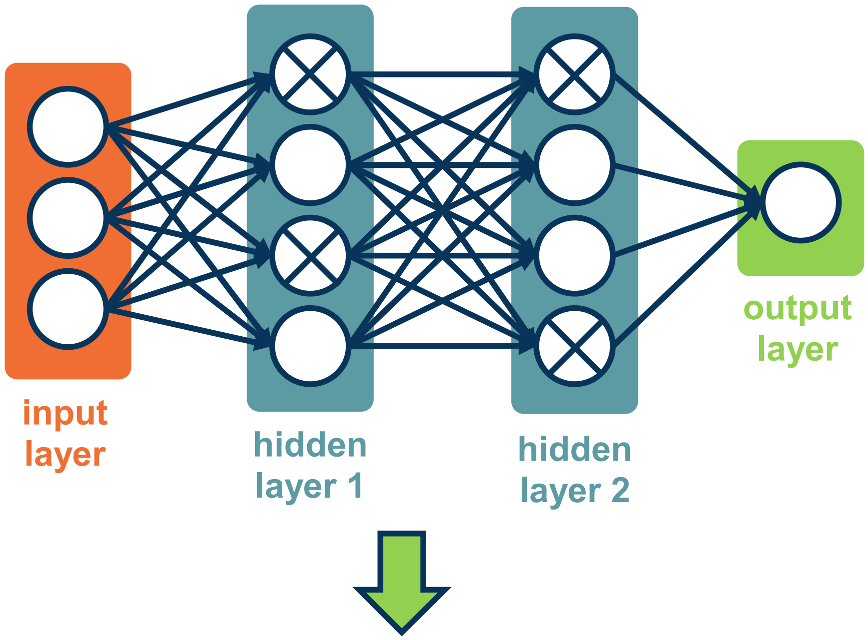
From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Dropout Implementation

- During training, each node has an expected $p * fan\_in$ nodes
- During test all nodes are activated
- **Principle:** Always try to have similar train and test-time input/output distributions!

Solution: During test time, scale outputs (or equivalently weights) by p

- i.e. $W_{test} = pW$
- Alternative: Scale by $\frac{1}{p}$ at train time



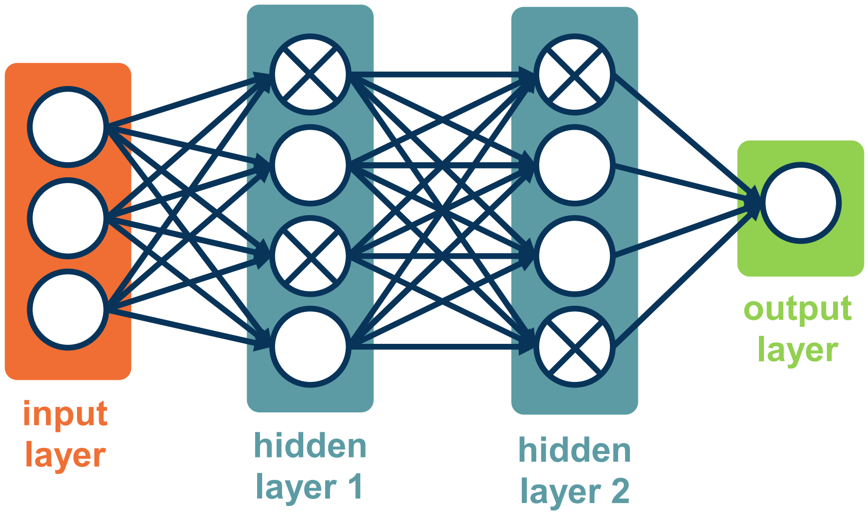
$$\begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ a_{41} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

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

Inference with Dropout

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

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



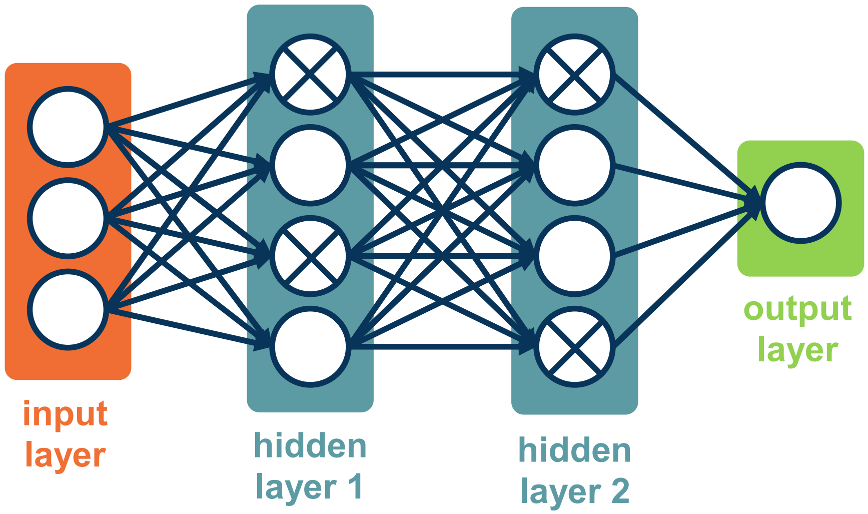
From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

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



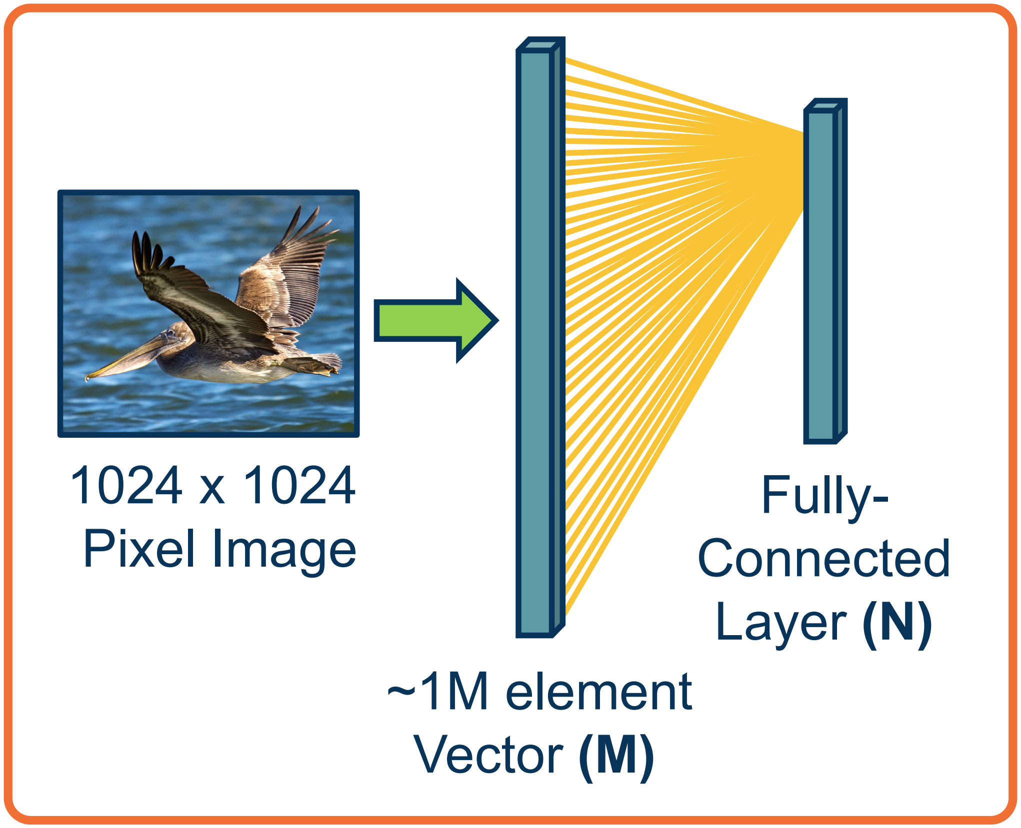
From: Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Srivastava et al.

Why Dropout Works

Convolution & Pooling

Georgia Tech

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?

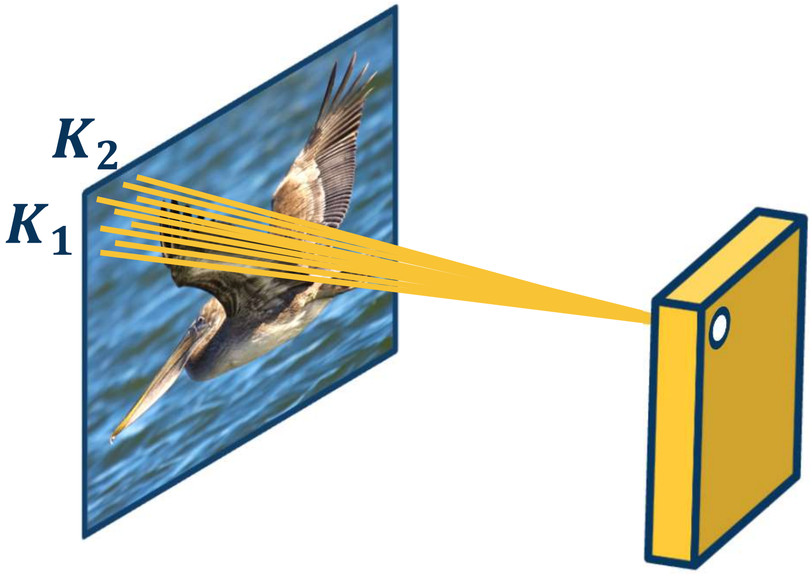
Limitation of Linear Layers

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)

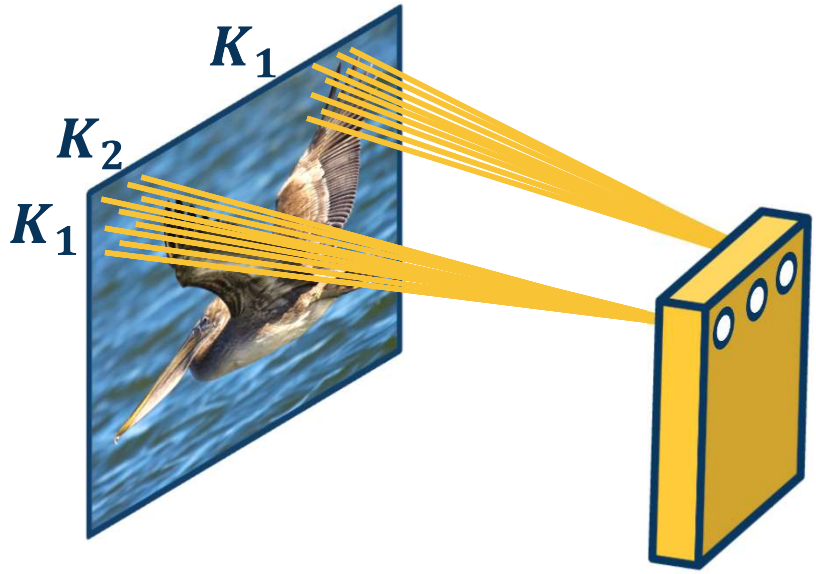
- Region from which a node receives input from is called its **receptive field**

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



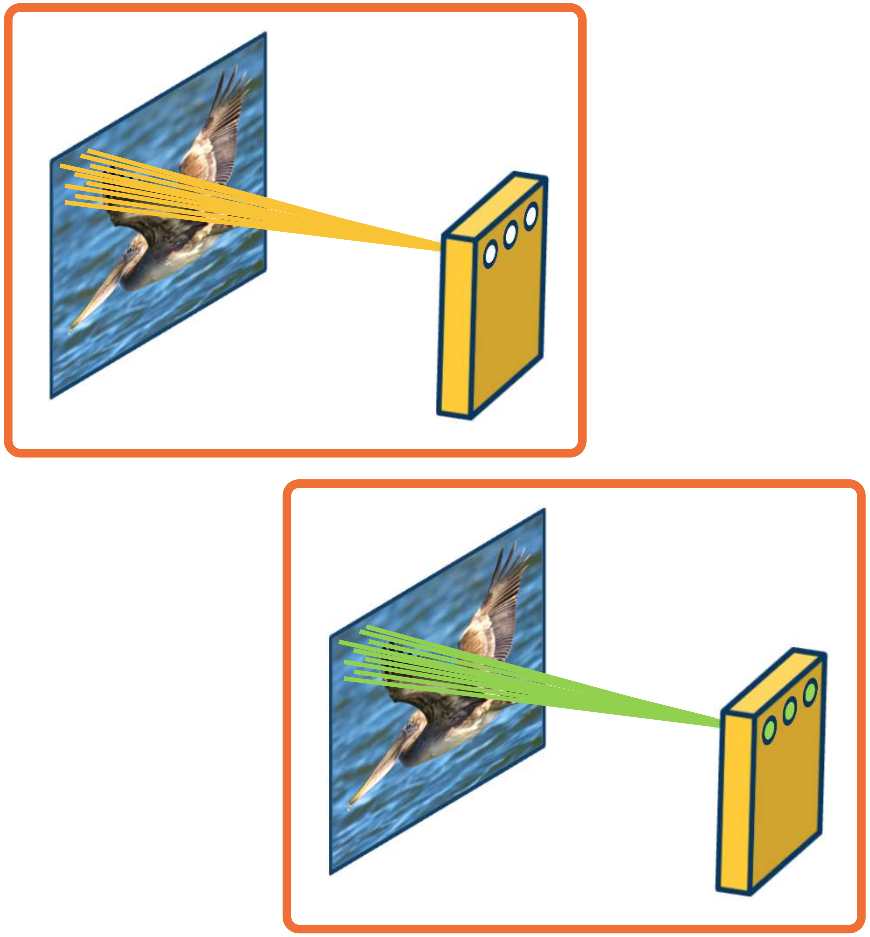
Nodes in different locations can **share** features

- 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

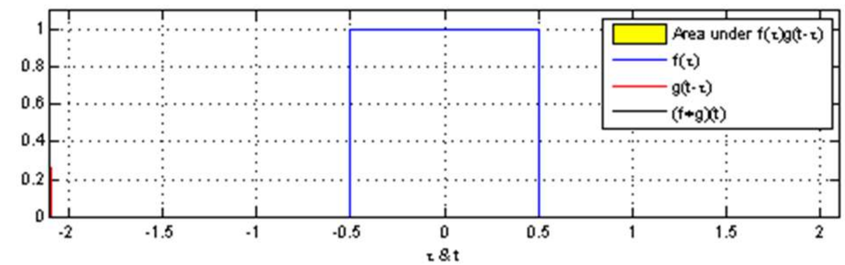


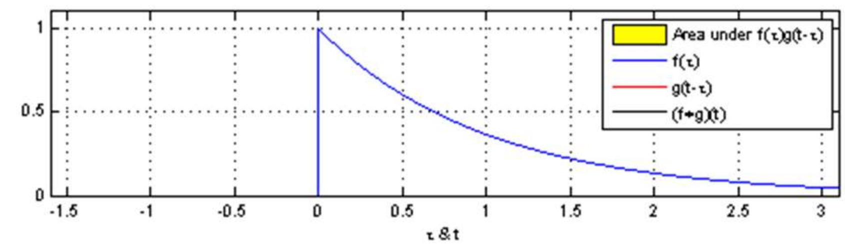
We can learn **many** such features for this one layer

- 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!





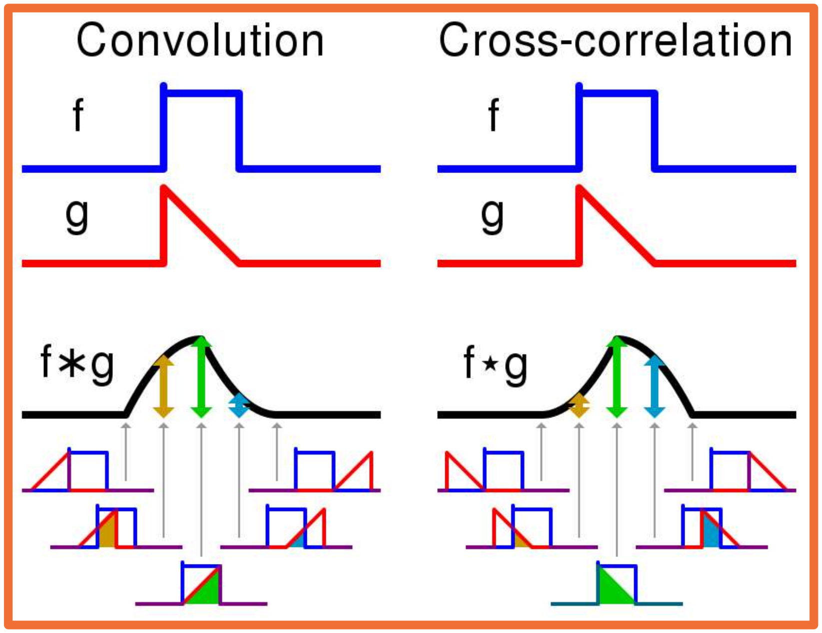
From https://en.wikipedia.org/wiki/Convolution

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

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

2D Discrete 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

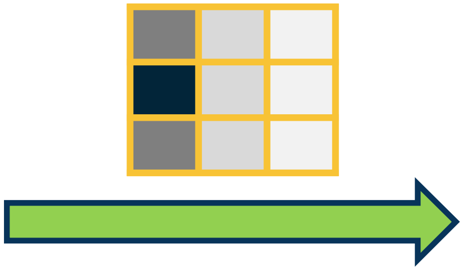
$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

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

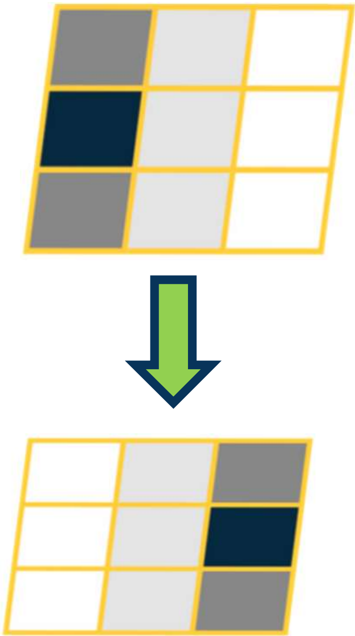
Kernel
(or filter)

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

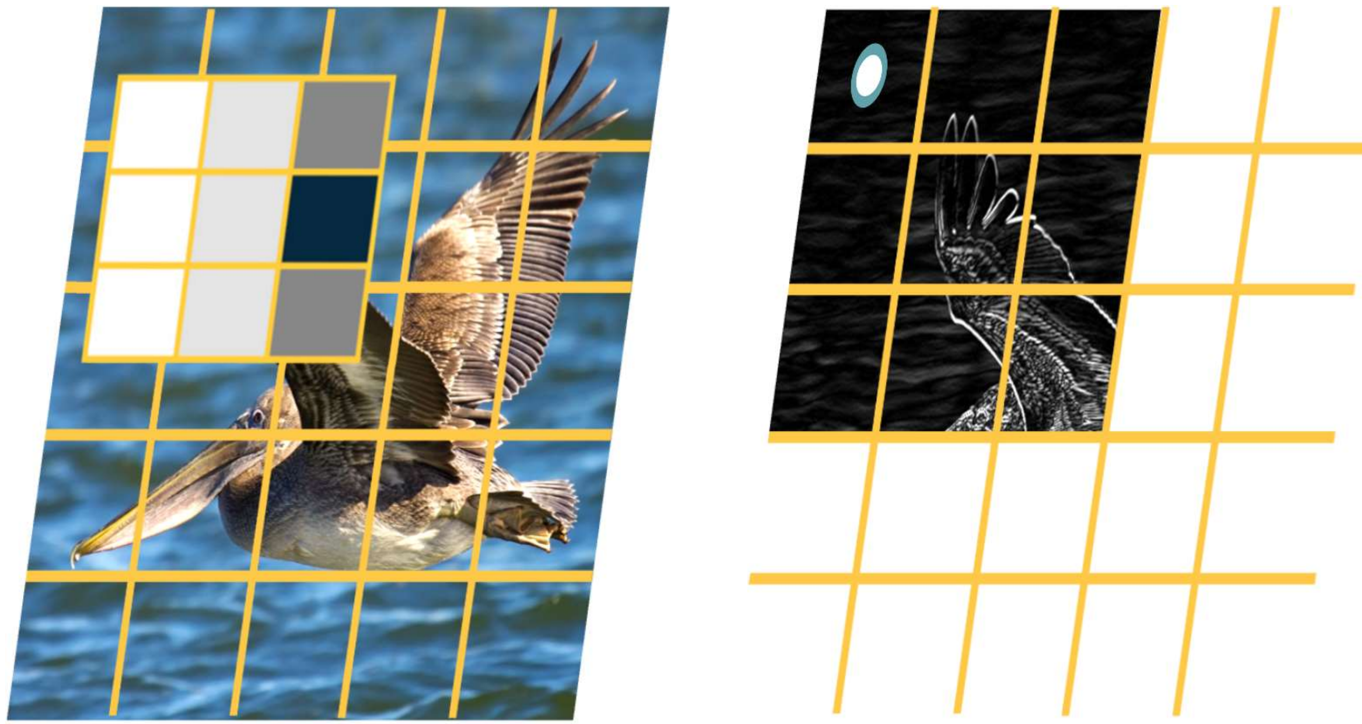
Output / filter / feature map

2D Discrete Convolution

1. Flip kernel (rotate 180 degrees)

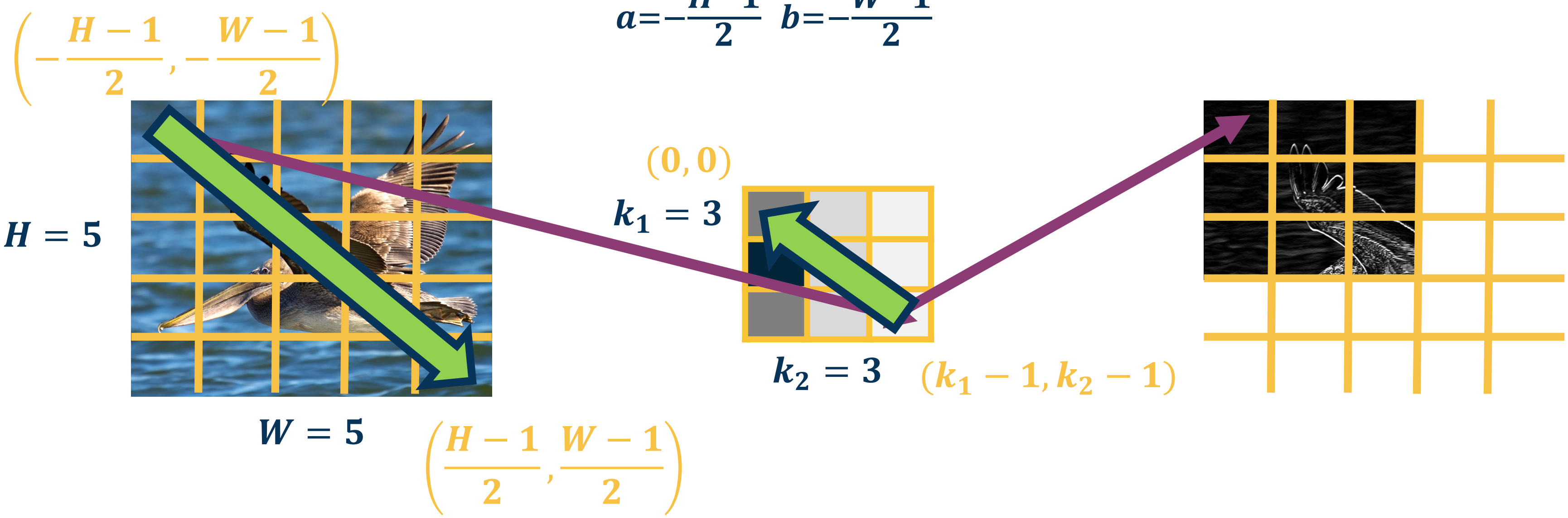


2. Stride along image



The Intuitive Explanation

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



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

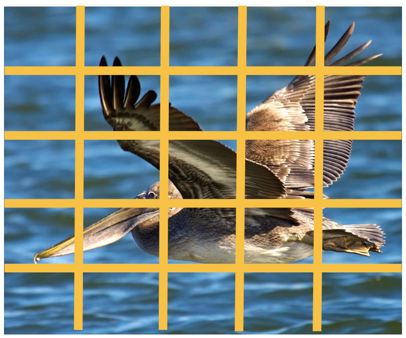
Mathematics of Discrete 2D Convolution

Georgia Tech

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

$(0,0)$

$H = 5$

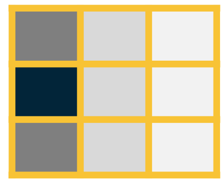


$W = 5$

$(H-1, W-1)$

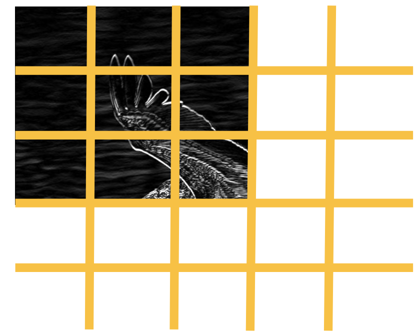
$\left(-\frac{k_1-1}{2}, -\frac{k_2-1}{2}\right)$

$k_1 = 3$



$k_2 = 3$

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



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)

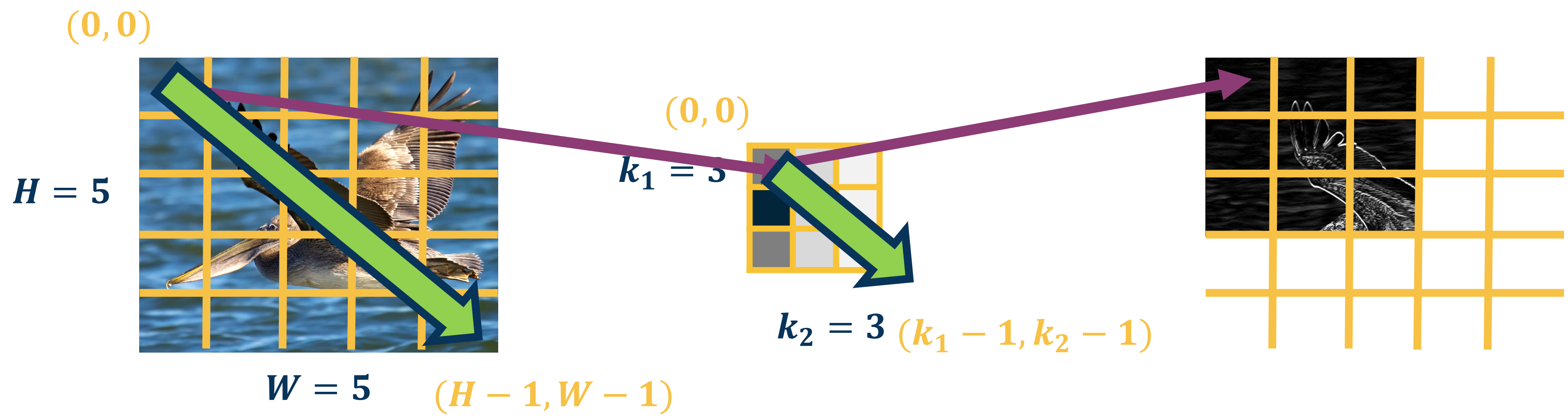
An **intuitive interpretation** of the relationship:

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

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

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

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



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

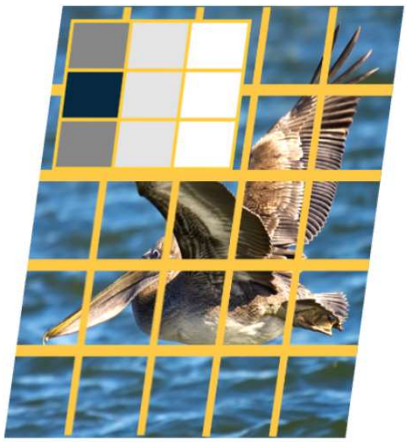
Cross-Correlation

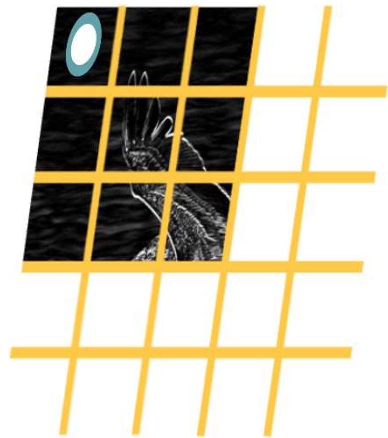
$$X(0:2,0:2) = \begin{bmatrix} 200 & 150 & 150 \\ 100 & 50 & 100 \\ 25 & 25 & 10 \end{bmatrix}$$

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

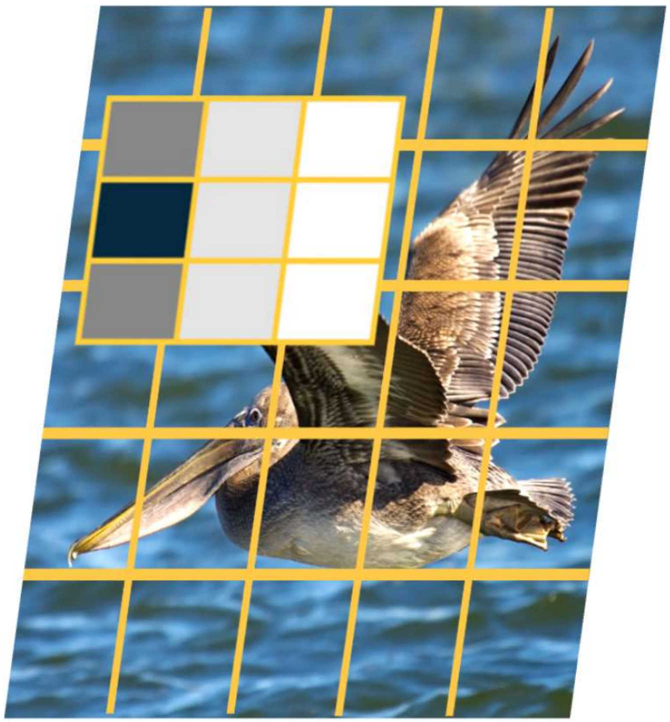
$$X(0:2,0:2) \cdot K' = 65 + \text{bias}$$

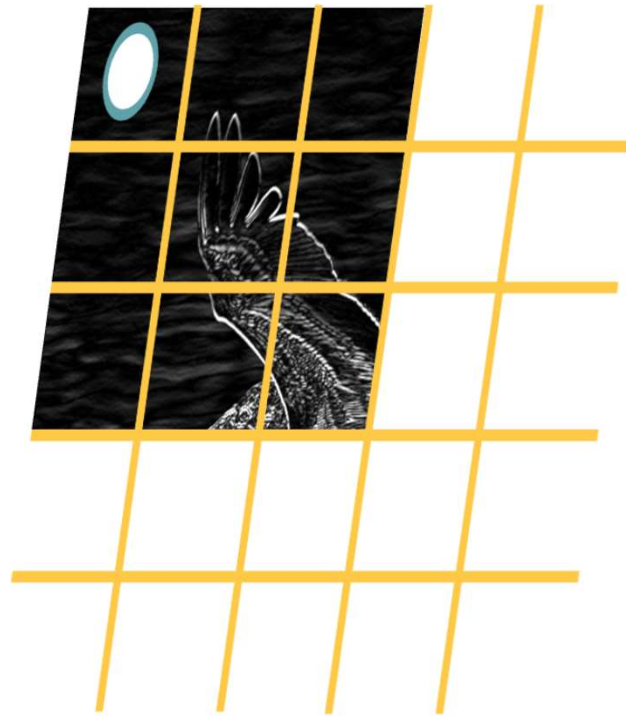
Dot product
(element-wise multiply and sum)



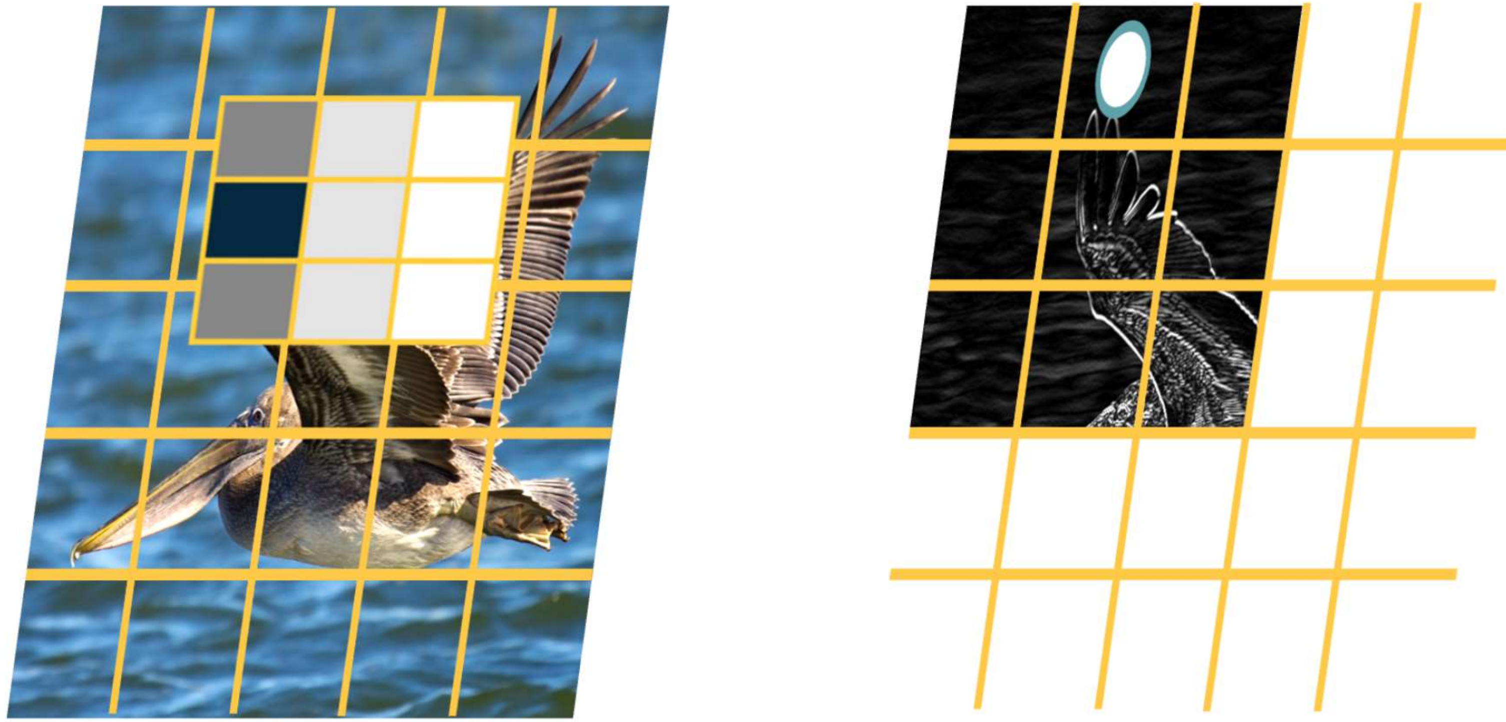


Cross-Correlation

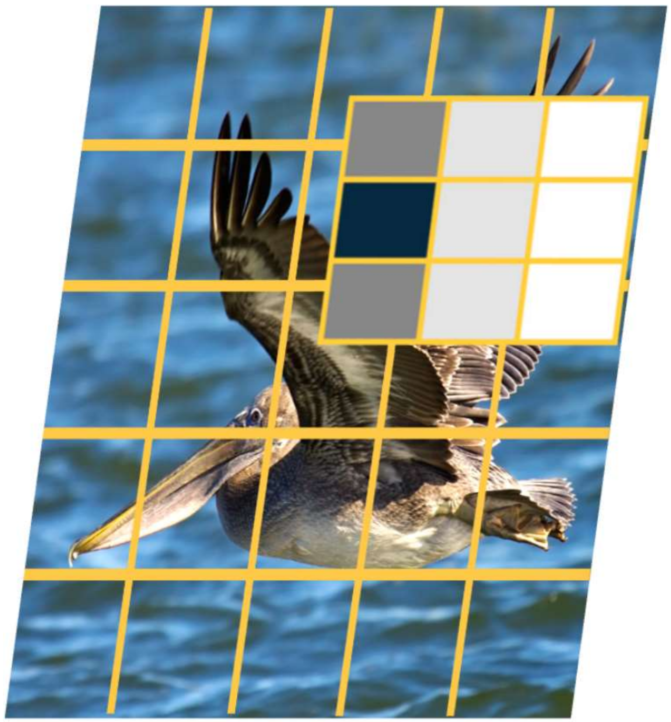


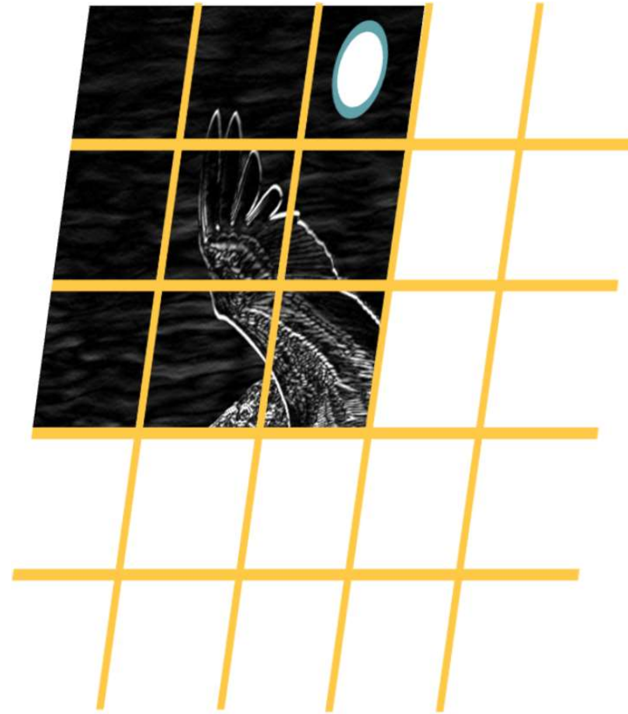


Convolution and Cross-Correlation

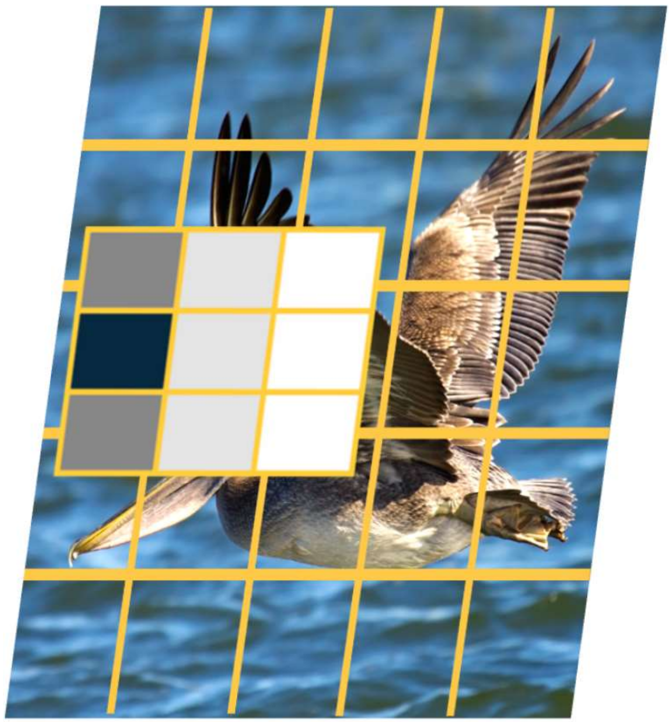


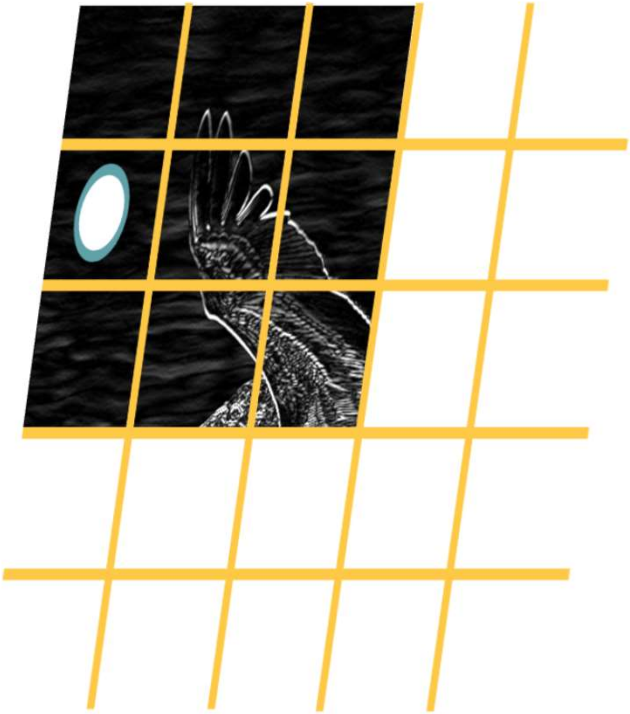
Convolution and Cross-Correlation



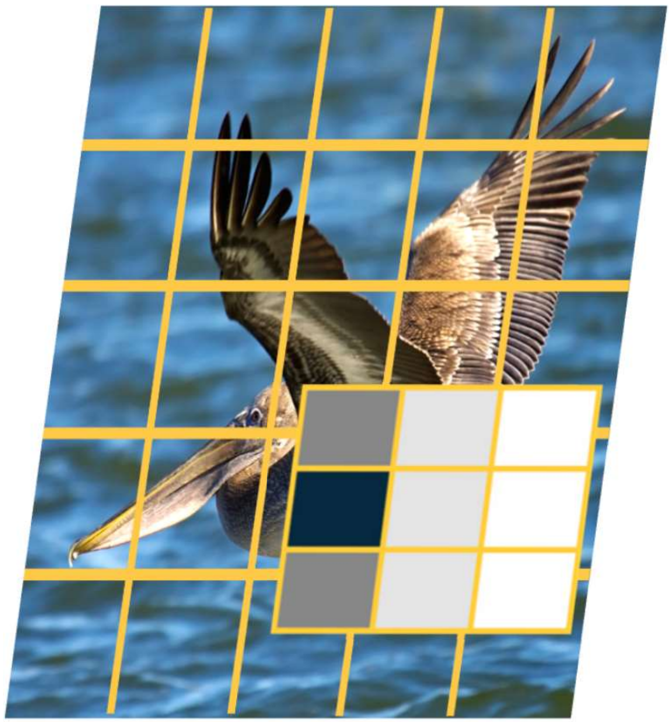


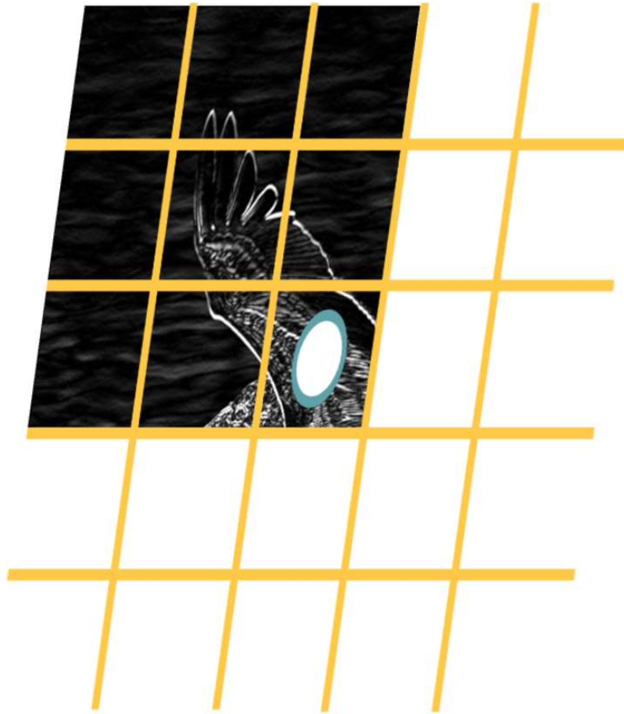
Convolution and Cross-Correlation





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

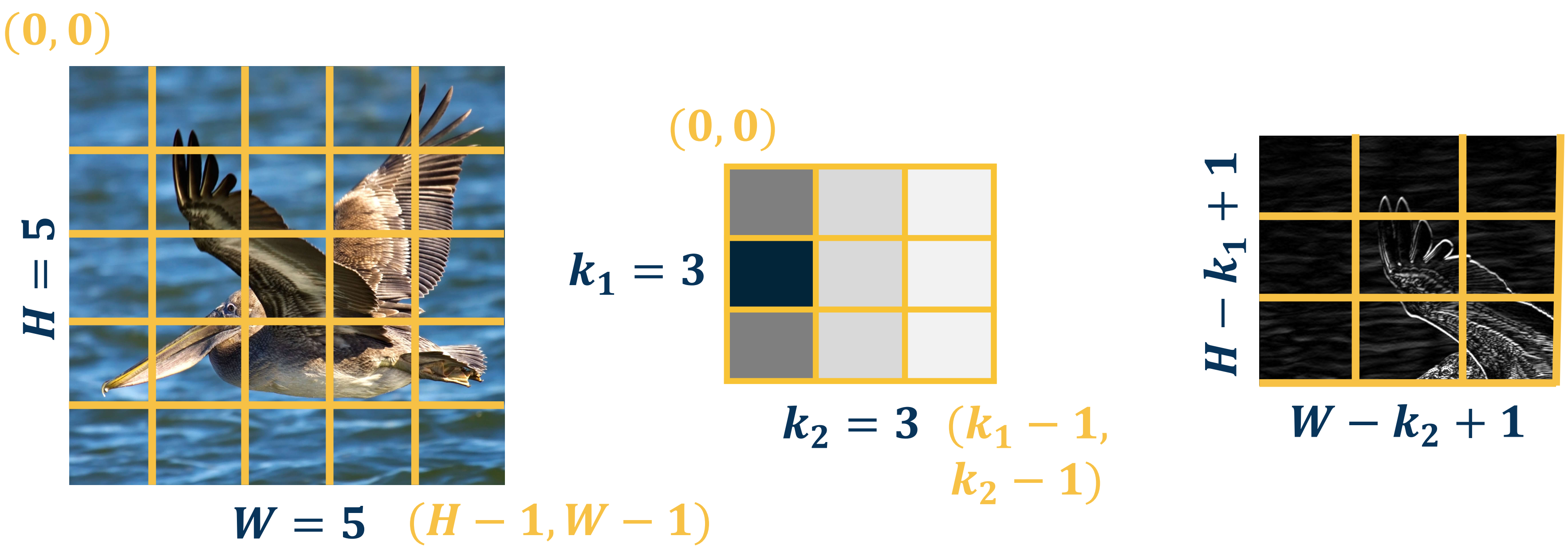
- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding**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

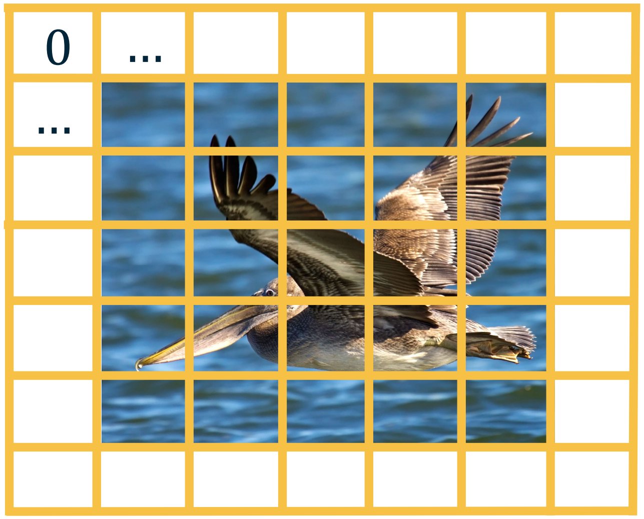


Valid Convolution

Adding Padding

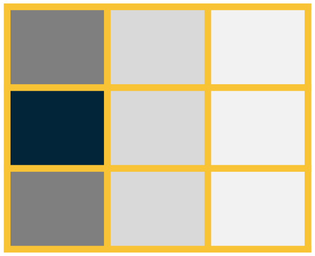
We can **pad the images** to make the output the same size:

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



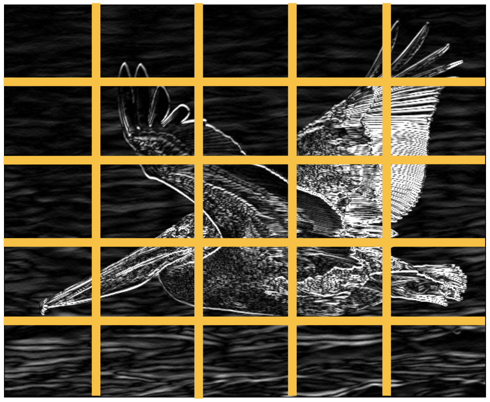
$H + 2$

$W + 2$



k_1

k_2



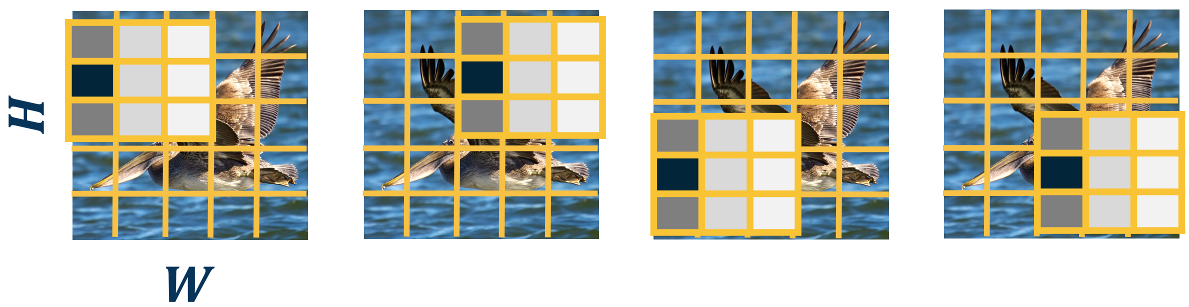
$H + 2 - k_1 + 1$

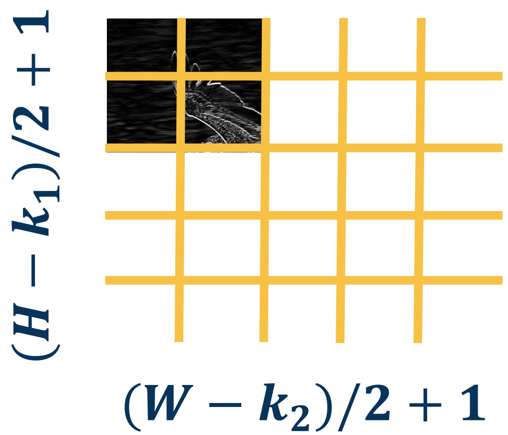
$W + 2 - k_2 + 1$

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)



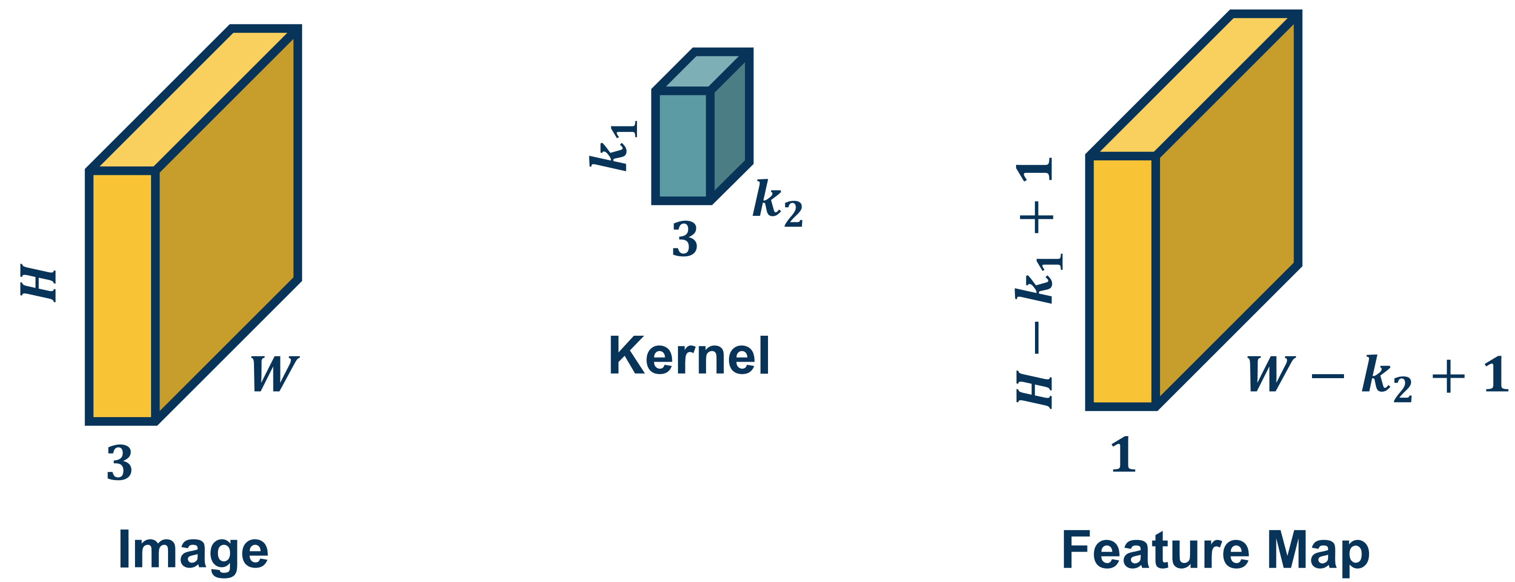


Stride can result in **skipped pixels**, e.g. stride of 3 for 5x5 input

Invalid Stride

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

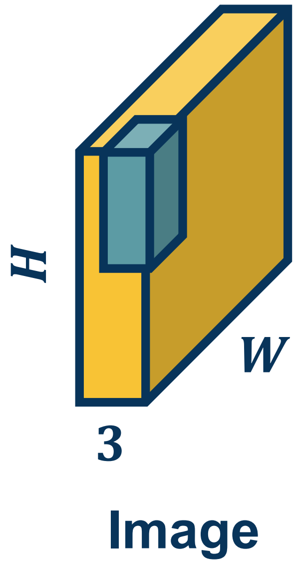
- In such cases, we have **3-channel kernels**!



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**!



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

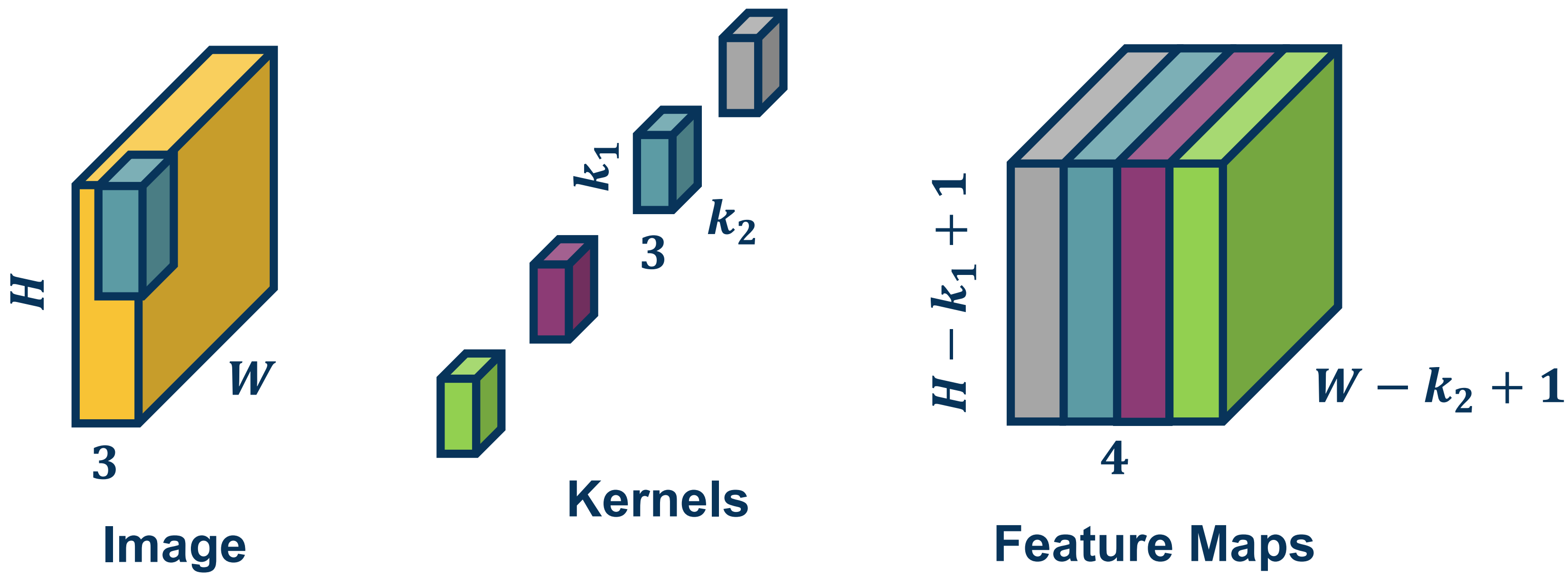
- Except with $k_1 * k_2 * 3$ values

Operation of Multi-Channel Input

We can have **multiple kernels per layer**

- We stack the feature maps together at the output

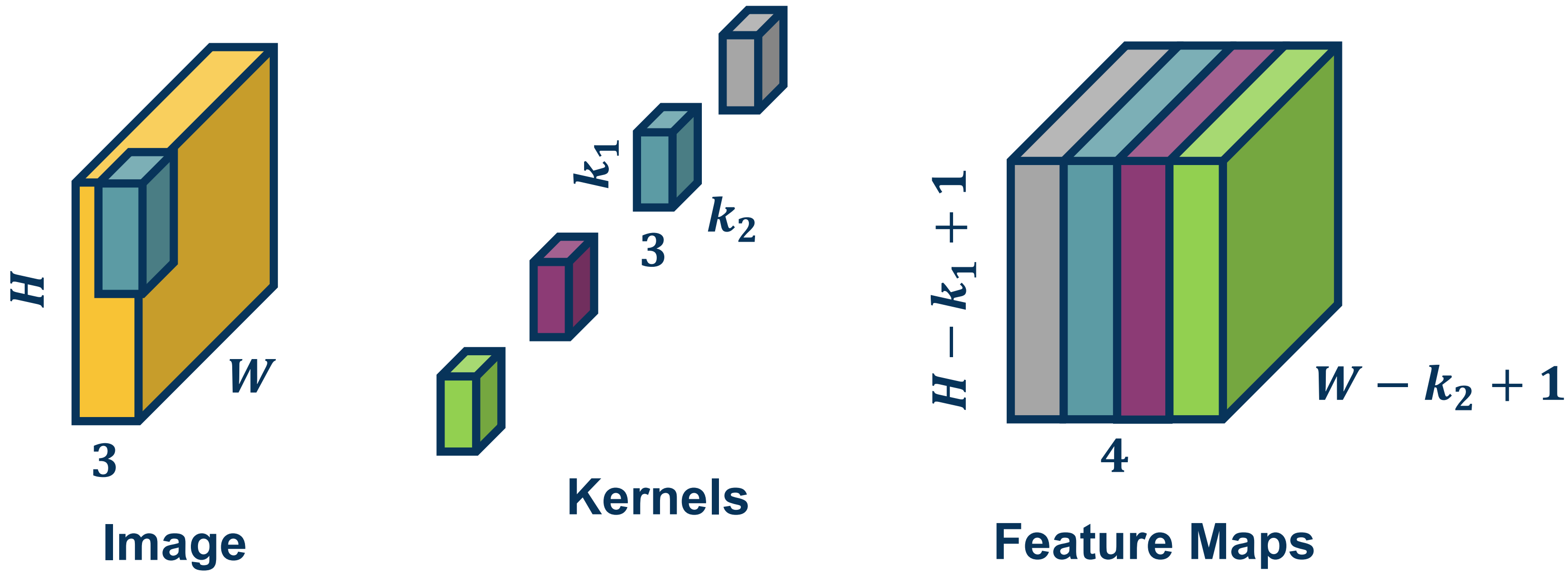
Number of channels in output is equal to *number of kernels*



Multiple Kernels

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

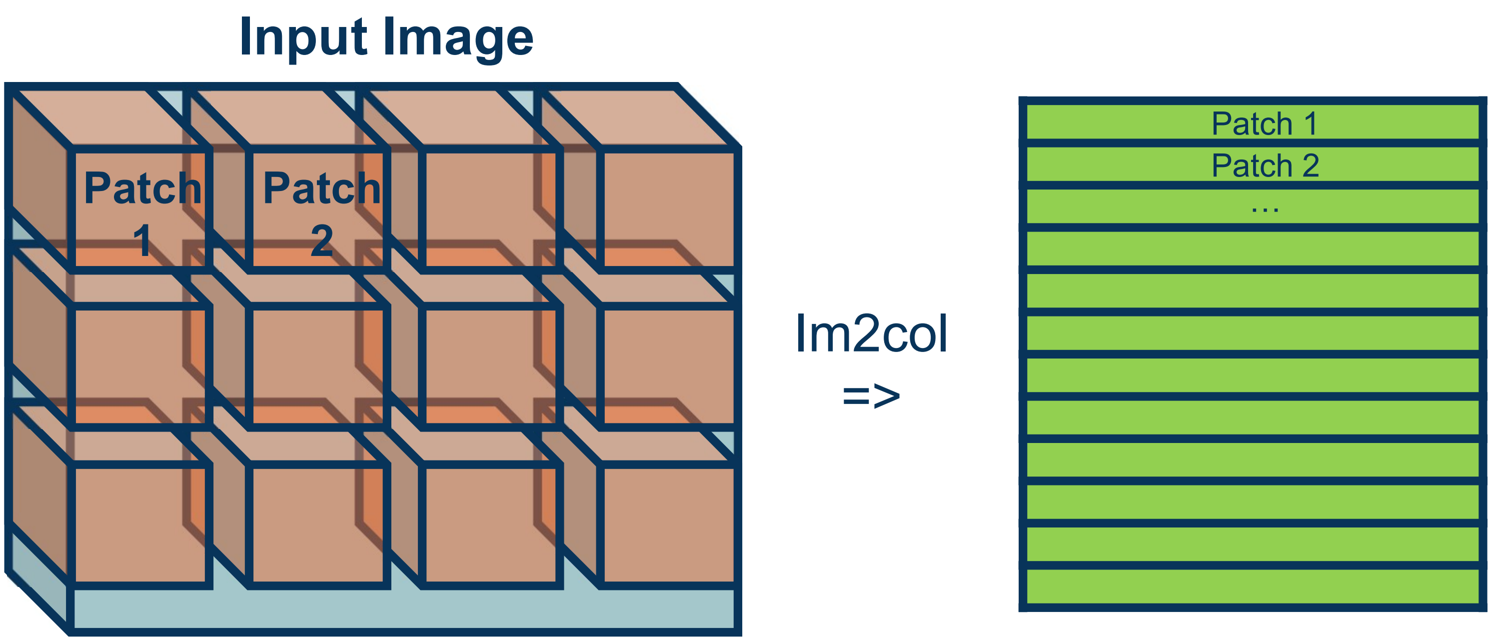
- Example:
  $k_1 = 3, k_2 = 3,\ N = 4\ input\ channels = 3$, then $(3 * 3 * 3 + 1) * 4 = 112$



Number of Parameters

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

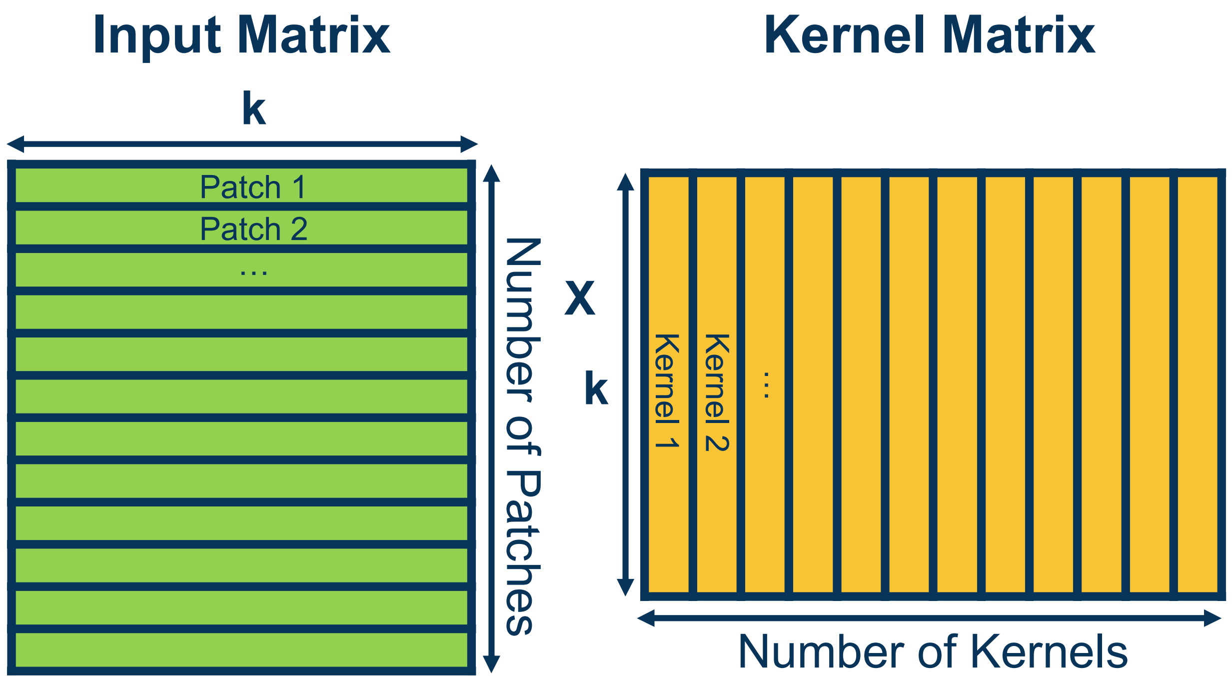
- **Step 1:** Lay out image patches in vector form (note can overlap!)



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/