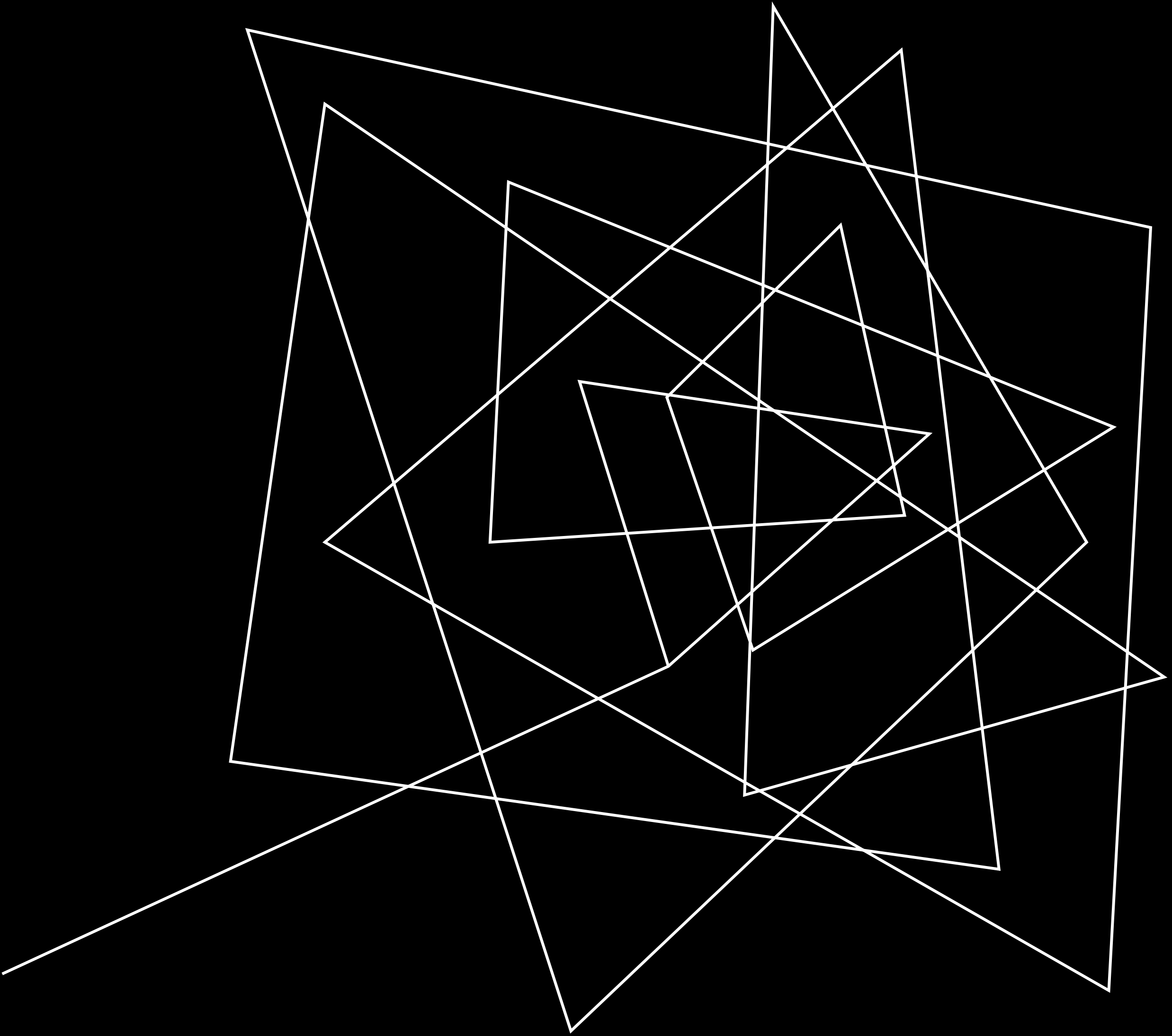
INTRODUCTION TO CONVOLUTIONAL NEURAL NETWORKS (CNN)

BY
SAIFUL BARI IFTU
LECTURER, DEPT. OF CSE, BRAC UNIVERSITY

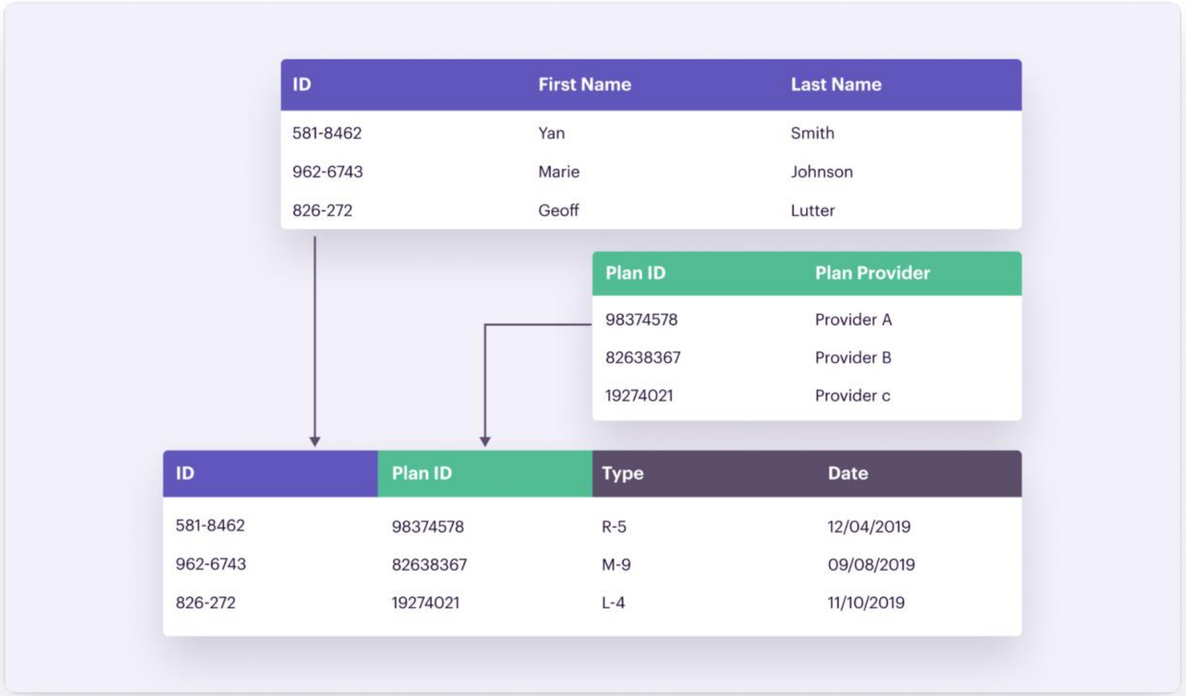
CONTENTS



UNSTRUCTURED DATA: IMAGES & VIDEOS

STRUCTURED DATA vs. UNSTRUCTURED DATA

Structured data resides in relational databases, which are structured to recognize relations between stored items of data. Databases of this type are typically managed via a relational database management system ("RDBMS"). This is usually what people think of when they think of a database, i.e., a table of rows and columns containing related information.



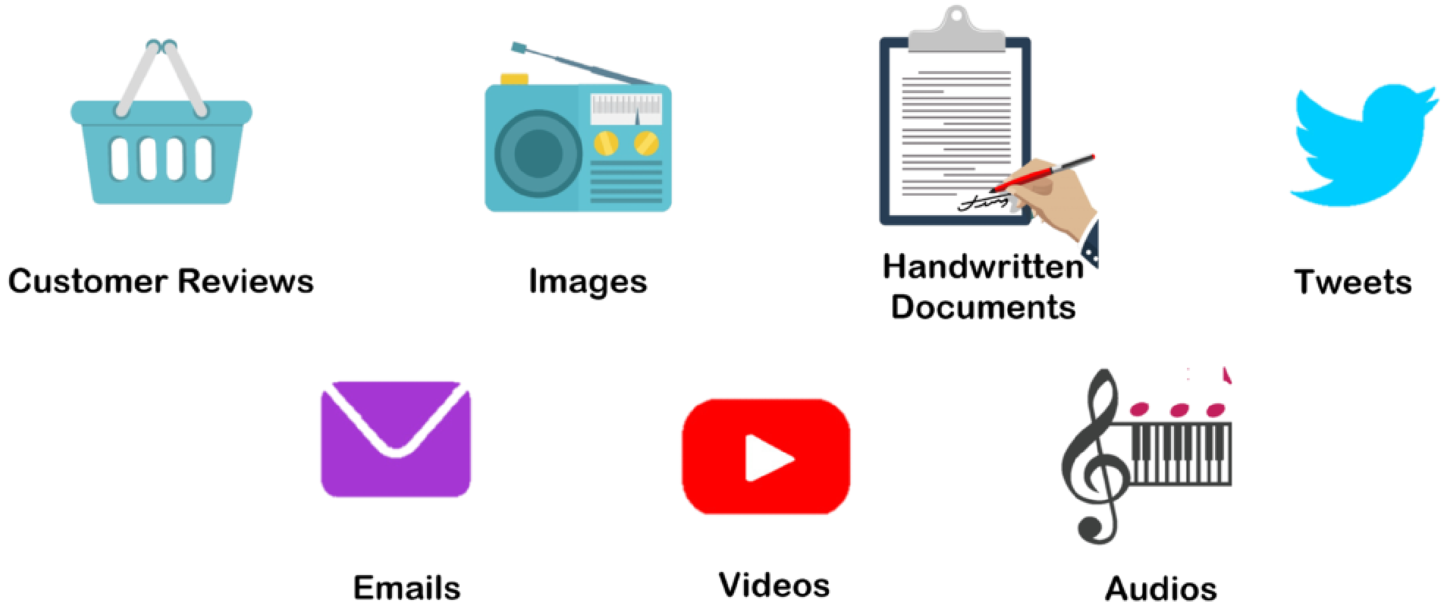
ID	First Name	Last Name
581-8462	Yan	Smith
962-6743	Marie	Johnson
826-272	Geoff	Lutter

Plan ID	Plan Provider
98374578	Provider A
82638367	Provider B
19274021	Provider c

ID	Plan ID	Type	Date
581-8462	98374578	R-5	12/04/2019
962-6743	82638367	M-9	09/08/2019
826-272	19274021	L-4	11/10/2019

STRUCTURED DATA vs. UNSTRUCTURED DATA

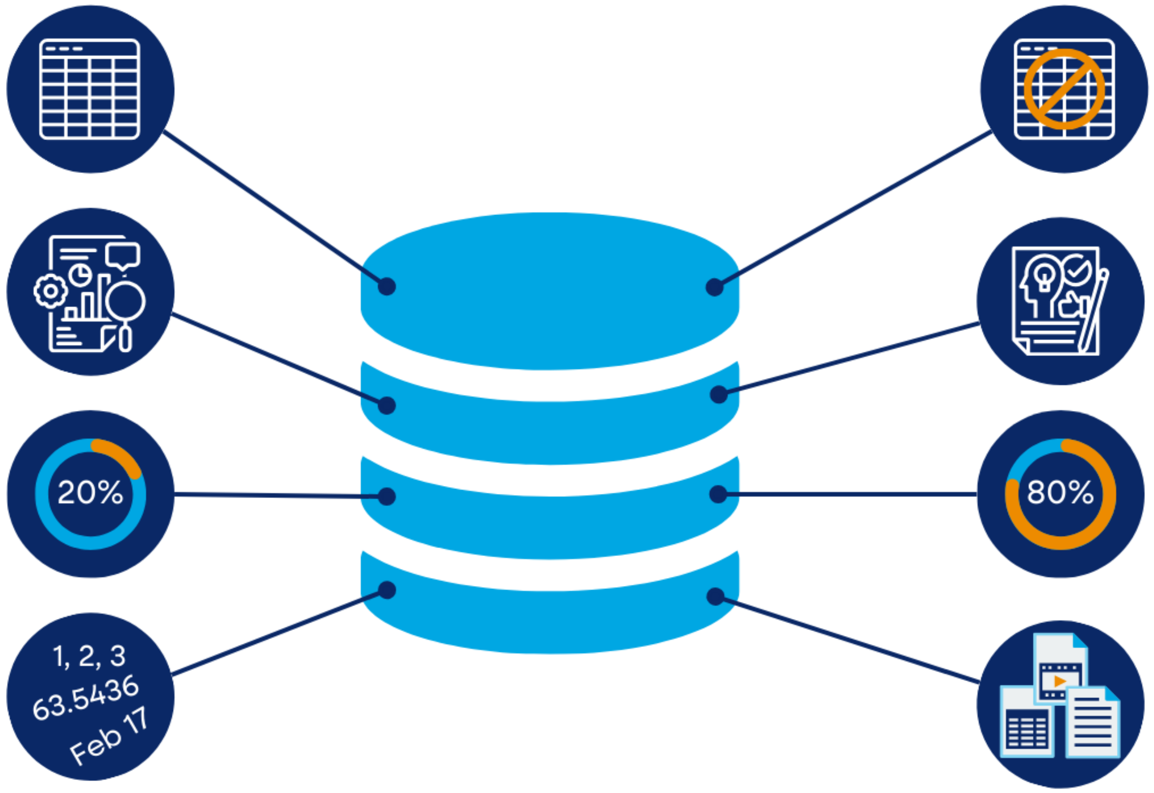
Unstructured data lacks a predefined format or structure. Unstructured data is everything else other than structured data. It has an internal structure (i.e., bits and bytes) but is not structured via pre-defined data models or schema, i.e., not organized and labeled to identify meaningful relationships between data. It may be textual / non-textual. It may be human / machine-generated. It might also be stored within a non-relational database like NoSQL. **Images/Videos are also examples of unstructured data.**



Structured Data vs Unstructured Data

Structured Data
Information that is highly organized, factual, and to-the-point

Unstructured Data
Doesn't have any predefined structure to it and comes in all its diversity of forms



Structured Data	Unstructured Data
Can be displayed in rows, columns, and relational databases	Cannot be displayed in rows, columns, or relational databases
Quantitative	Qualitative
Estimated 20% of enterprise data (Gartner)	Estimated 80% of enterprise data (Gartner)
Numbers, dates, and strings	Images, audio, video, word processing files, e-mails, spreadsheets

Source: https://prime46.com/structured-and-unstructured-data-for-strategic-decision-making/

IMAGES & VIDEOS

Structured data (e.g., relational databases) is neatly organized into tables with rows and columns. **Images and videos do not fit into this tabular format; they are stored as pixel grids (arrays), making them unstructured**.

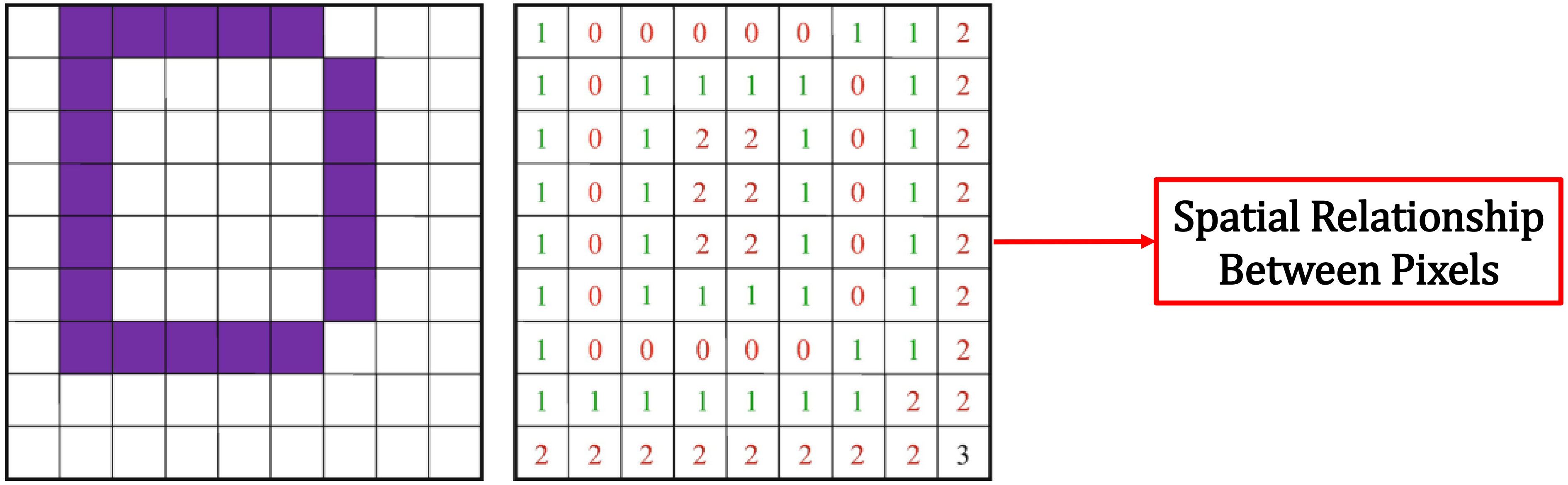
Images and Videos are characterized by their **High Dimensionality** and **Complexity**. An RGB image of 224×224 pixels has 150,528 values (224×224×3). A 1-minute HD video at 30 FPS has 1,800 frames, each being an image, exponentially increasing data size. Unlike structured data, these do not have explicit "features" like age or price; they require feature extraction.

Perhaps more importantly, Images and Videos exhibit **Spatial and Temporal Dependencies.** In images, Nearby pixels are related (e.g., edges and textures). The meaning of a pixel highly depends on its neighbors. In videos, Frames have a temporal relationship (motion and context over time).

IMAGES & VIDEOS

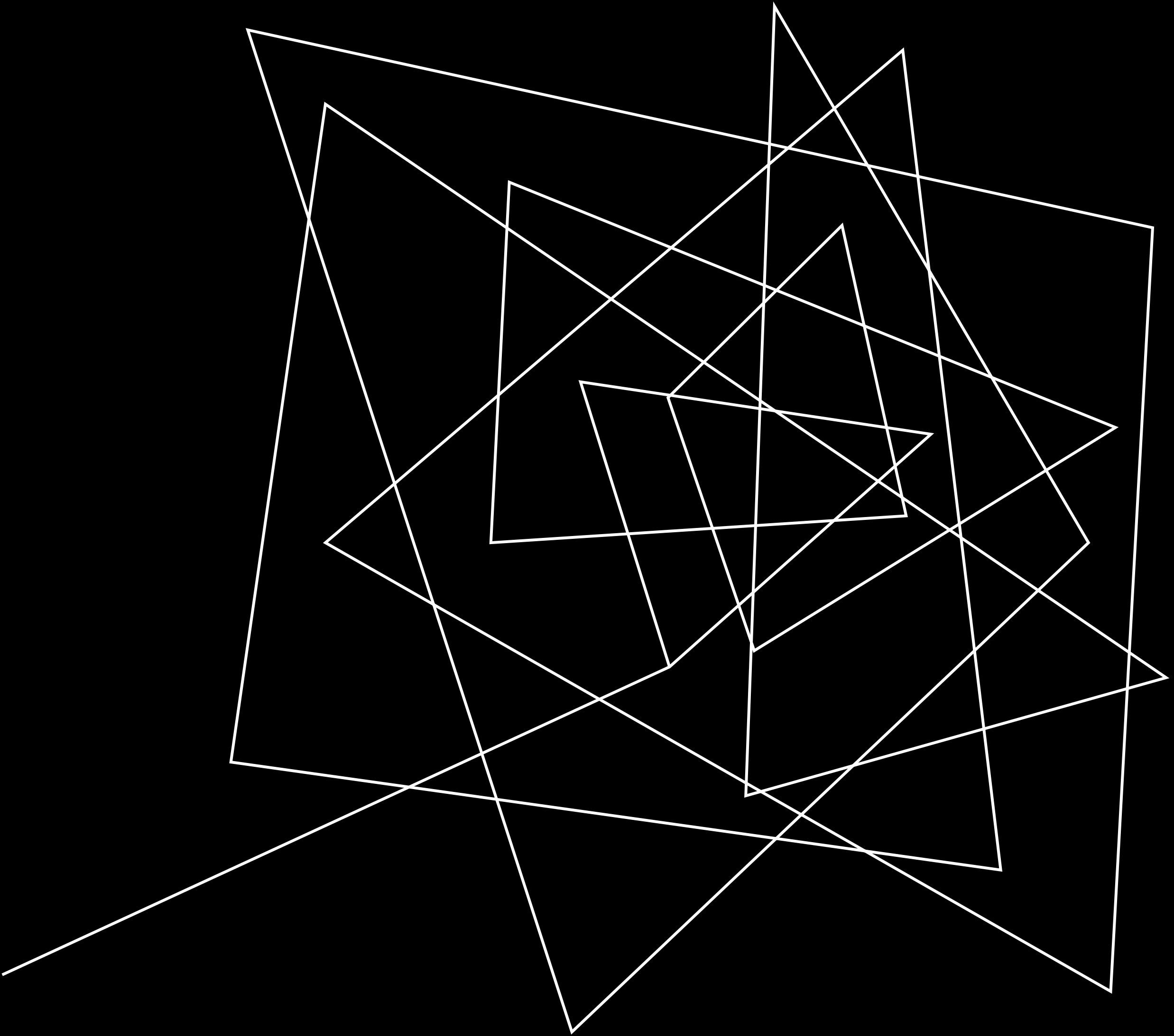
One more important aspect is that the pixel values in images/videos are **not directly interpretable**. In structured data, a column like salary = $50,000 has direct meaning. In an image, pixel values (e.g., 125, 200, 30 in RGB) do not directly describe objects. **The structure and meaning are implicit and need to be learned.**

All these factors point towards the need to adopt an approach that accommodates and learns the spatial and temporal dependencies within image/video data and does it in an efficient way!



1	0	0	0	0	0	1	1	2
1	0	1	1	1	1	0	1	2
1	0	1	2	2	1	0	1	2
1	0	1	2	2	1	0	1	2
1	0	1	2	2	1	0	1	2
1	0	1	1	1	1	0	1	2
1	0	0	0	0	0	1	1	2
1	1	1	1	1	1	1	2	2
2	2	2	2	2	2	2	2	3

Source: https://doi.org/10.1007/978-981-15-5873-3_9



CONVOLUTIONAL NEURAL NETWORKS (CNN): WHAT & WHY?

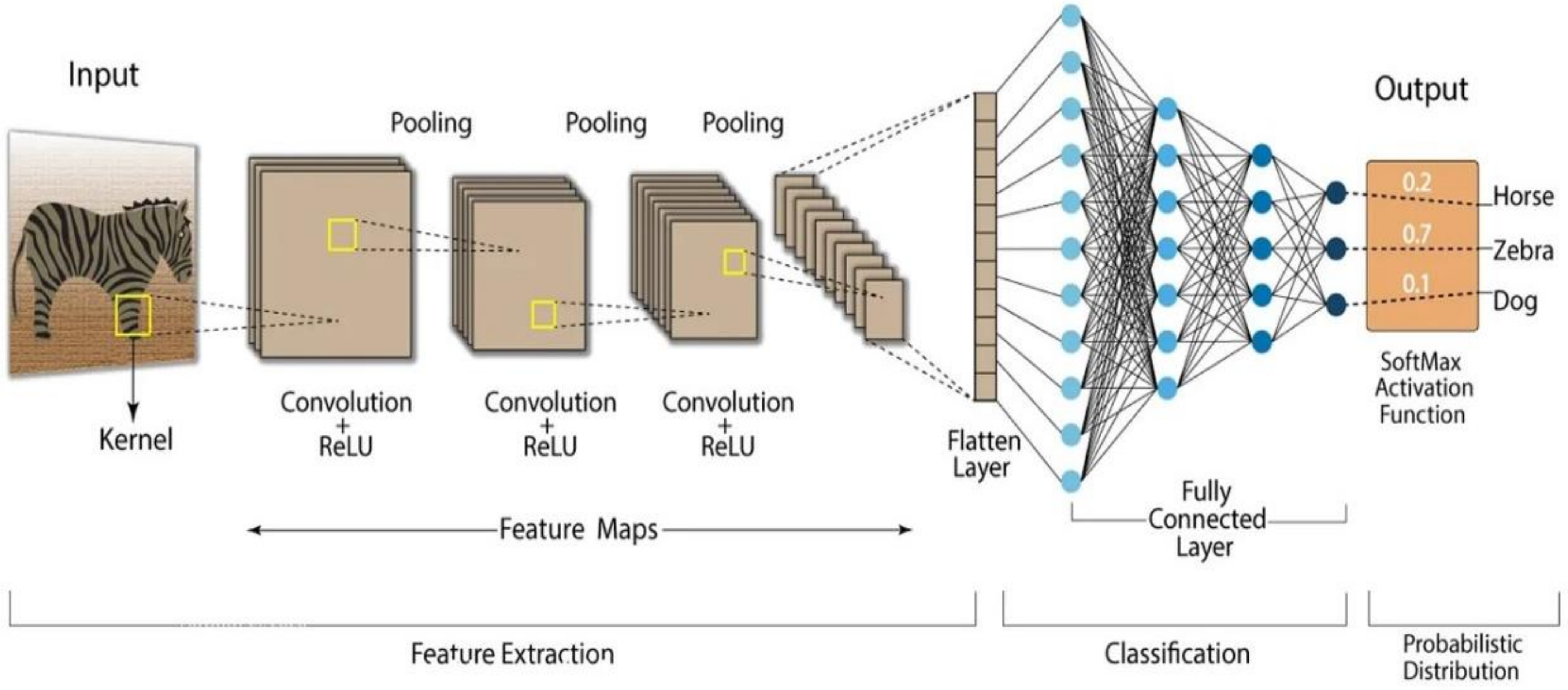
CNN

A **Convolutional Neural Network (CNN)** is a type of deep learning model specifically designed to process structured grid data, such as **images** and **videos**. CNNs automatically extract spatial features from images by using convolutional layers, reducing the need for manual feature engineering.

Key Principles Behind CNNs

- **Local Connectivity:** Each neuron processes a small patch of the image rather than the entire image.
- **Weight Sharing:** A filter (kernel) is used across the entire image, reducing trainable parameters.
- **Hierarchical Feature Learning:**
  - ➢ Early layers detect edges and textures.
  - ➢ Mid layers detect shapes and patterns.
  - ➢ Deep layers recognize complex objects (faces, animals, etc.).

CNN

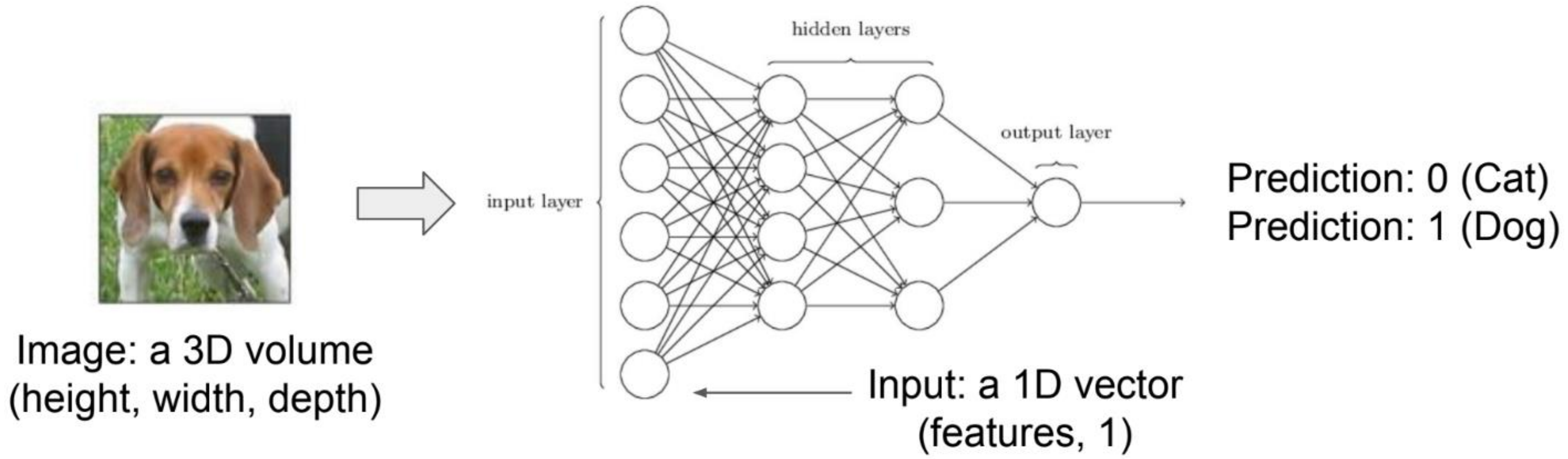


A Typical Convolutional Neural Network (CNN)

WHY NOT MLP?

For Computer Vision tasks (concerning images/videos), feed-forward fully connected neural networks (or MLPs) are not suitable for mainly two reasons:

- **A lack of spatial (and temporal for videos) reasoning**
- **An explosive number of parameters**



WHY CNN: COMPUTATIONAL EFFICIENCY

Fully Connected Neural Networks Struggle with High-Dimensional Data. For example, A **224×224** RGB image has **150,528** features. As a result, a fully connected layer would require millions of parameters (**150529** for each neuron in the first hidden layer),

This leads to:

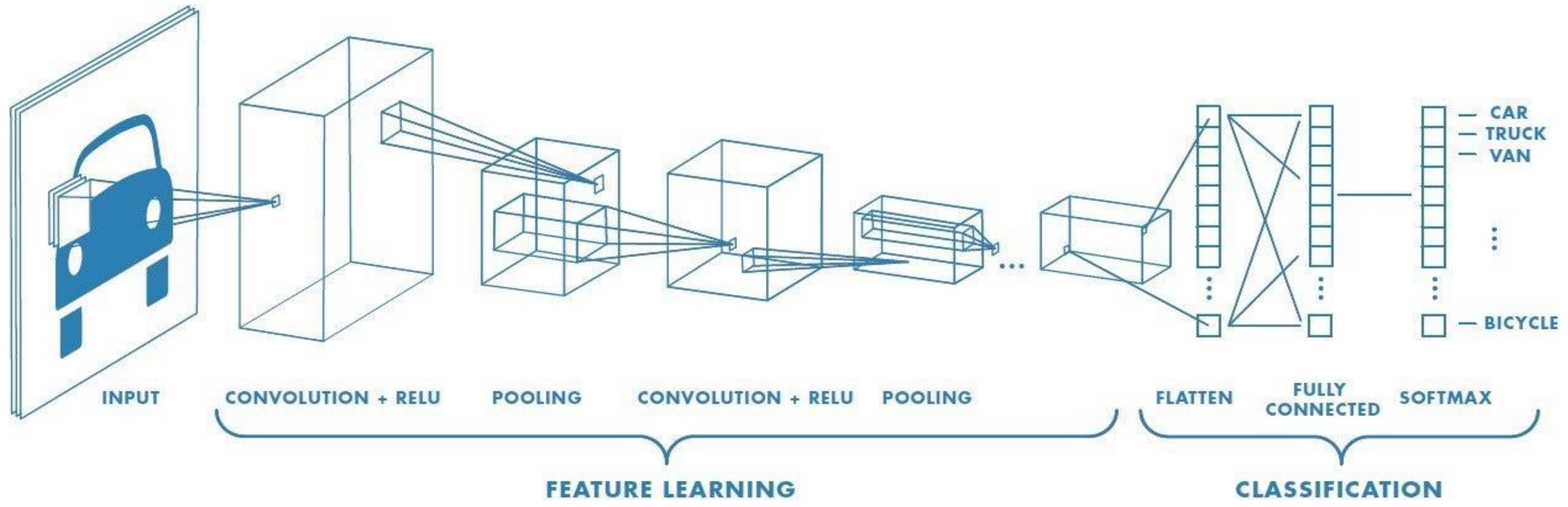
- Overfitting due to excessive trainable weights.
- High computational cost and memory requirements.

Computational Efficiency in CNNs: Convolutional filters are shared across the image, significantly reducing the number of trainable parameters. A **3×3** kernel has only **9** parameters (**27** parameters per filter in RGB), compared to thousands in a fully connected network. So, even if **100** filters are used, the parameter count will be just **900 (+100** for the bias term**)** for the first hidden layer. Compare that to an MLP with **100** neurons in the first hidden layer for the simple **28*28** MNIST dataset images. It would have **784×100 = 78,400** parameters.

WHY CNN: SPATIAL REASONING

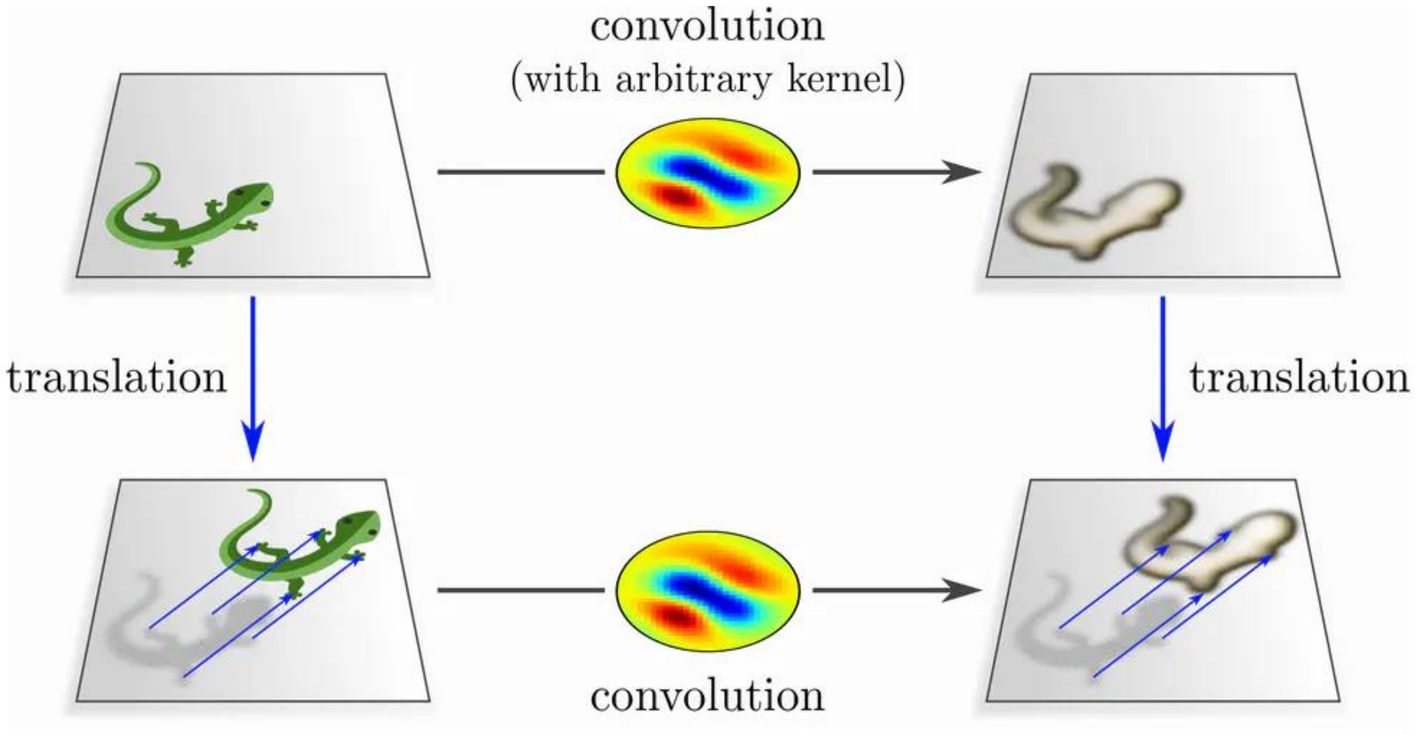
Fully Connected Neural Networks take the input as a 1D vector. So, it naturally has limited 2D/3D spatial or temporal reasoning ability. As a result, they perform poorly with images/videos.

Spatial Reasoning in CNNs: In CNNs, each output neuron connects to a small image patch (called the **receptive field**). So, they naturally have spatial awareness. The deeper layers have larger receptive fields, allowing the network to extract hierarchical features in different convolutional layers. Initial layers detect edges and textures, while deeper layers recognize shapes and objects.

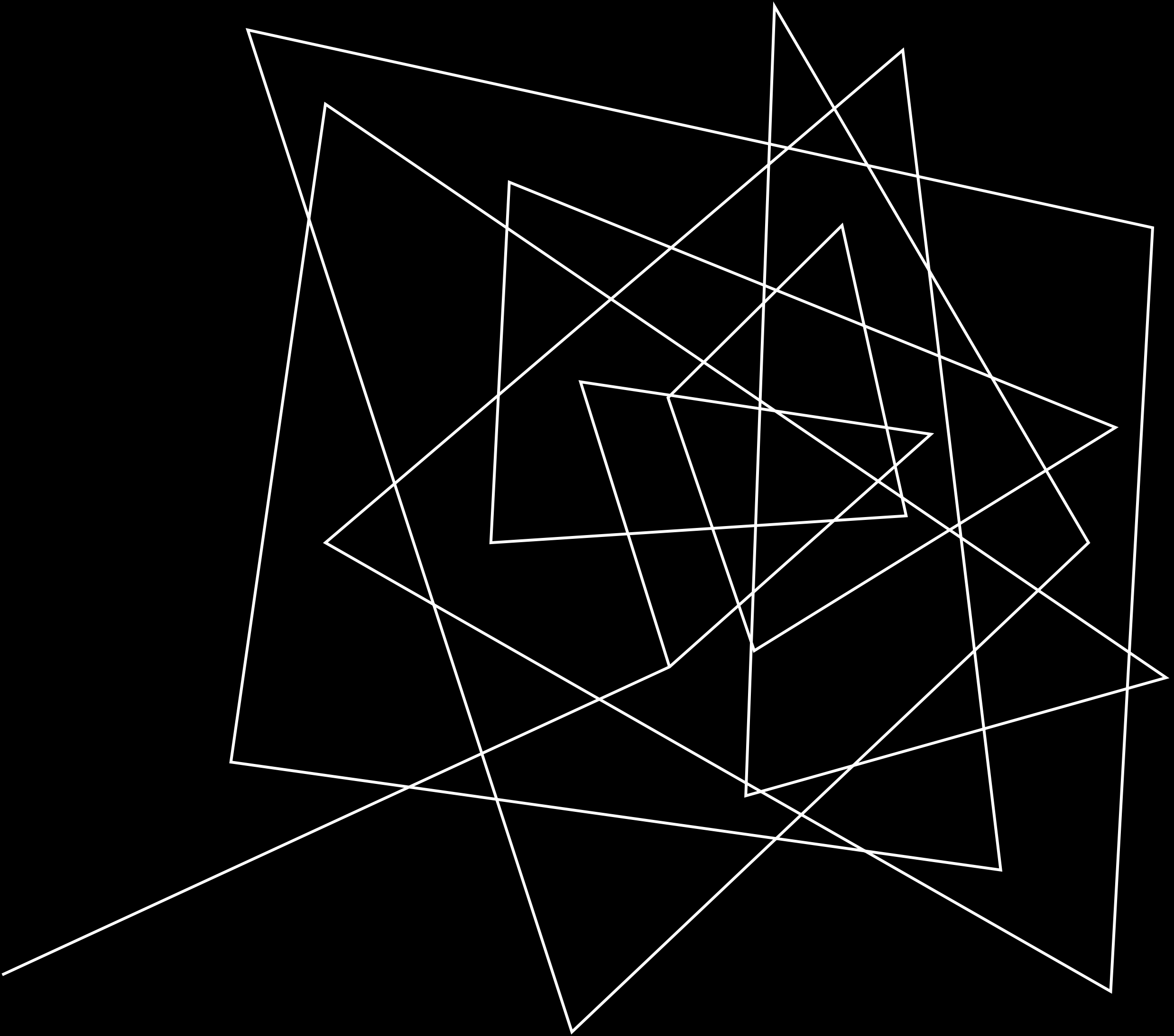


WHY CNN: TRANSLATIONAL INVARIANCE

Another aspect that 1D vectors from the Fully Connected Neural Networks struggle with is **translational invariance**. If the same object varies in position or scale within an image, MLPs can't detect it. However, in CNNs, each neuron in a convolutional layer looks at a small region of the image. As the filter slides (convolves) across the image, it checks all locations for that feature. This sliding nature ensures that features are detected anywhere they appear.



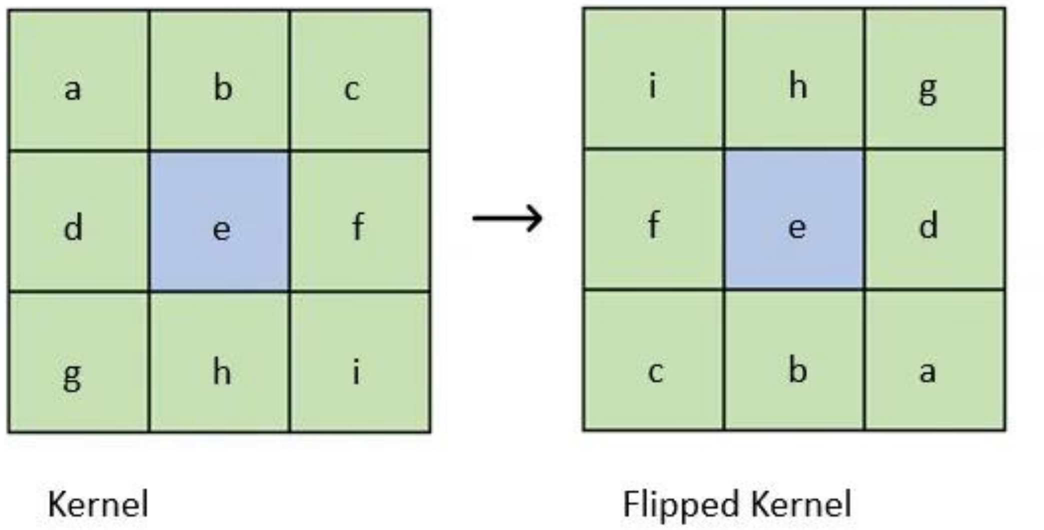
Source: https://maurice-weiler.gitlab.io/blog_post/cnn-book_2_conventional_cnns/



THE CONVOLUTION PROCESS

KERNELS

A **Kernel** is a small matrix used to apply effects like blurring, sharpening, edge detection, etc., on an image. It operates by sliding over the image, performing element-wise multiplication with the image pixels, and summing the results to produce a new pixel value. Kernels are also used in CNNs.



Source: https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b

THE CONVOLUTION PROCESS

Convolution is the process of sliding a filter (kernel) over an image, sequence, video, etc., and computing dot products (point-wise multiplication) to produce another similar signal (called a **feature map**, usually).

Convolution Operation Equation (2D Cross-Correlation Form)

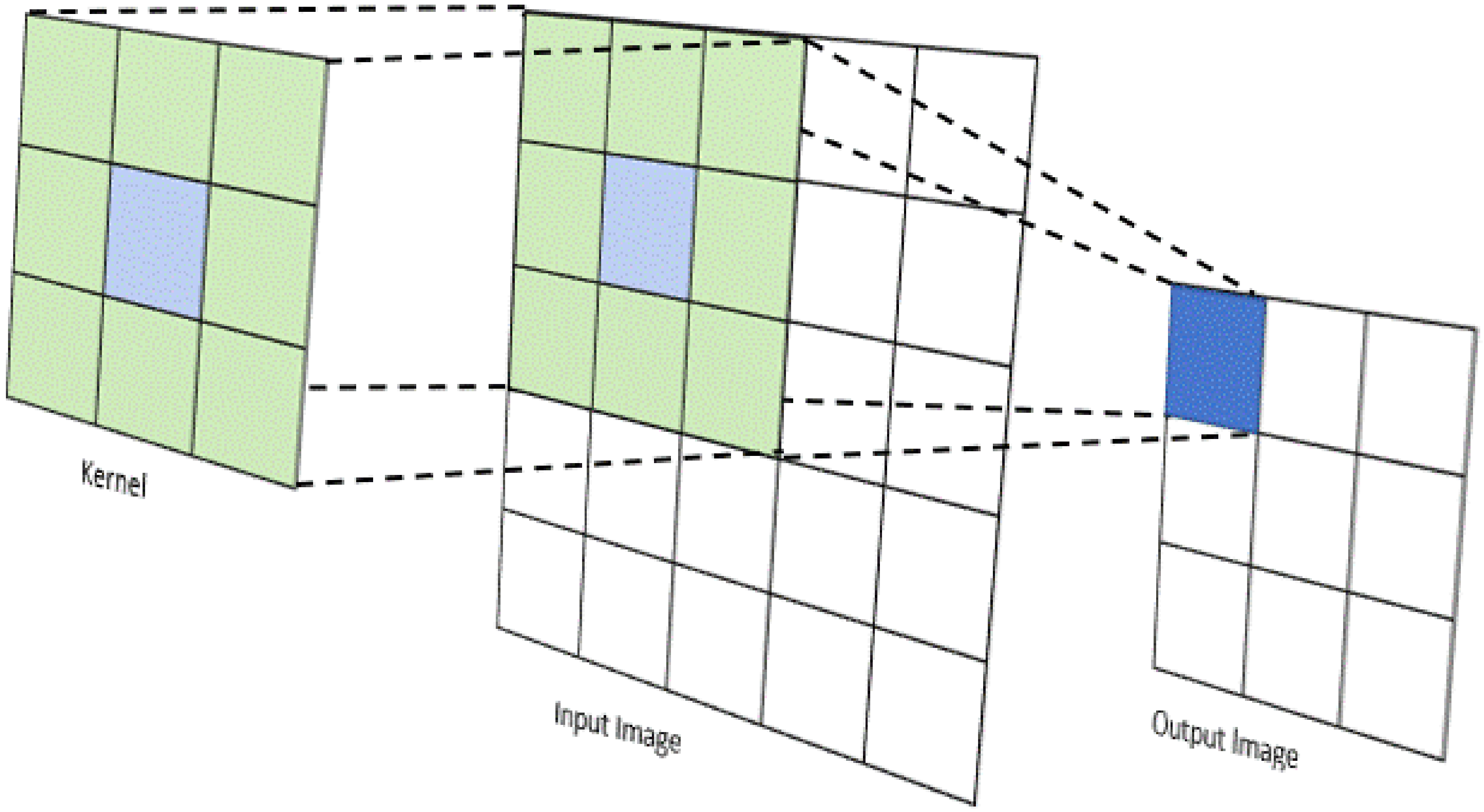
$$\boldsymbol{(I * K)_{i,j}} = \sum_{i=1}^{m} \sum_{j=1}^{n} \boldsymbol{I_{i+m,j+n} \bullet Km_{,n}}$$

Where:

- $S(i,j)$ is the output (feature map) at position (i,j),
- I is the input image,
- K is the kernel (filter),
- m,n iterate over the filter dimensions.

THE CONVOLUTION PROCESS

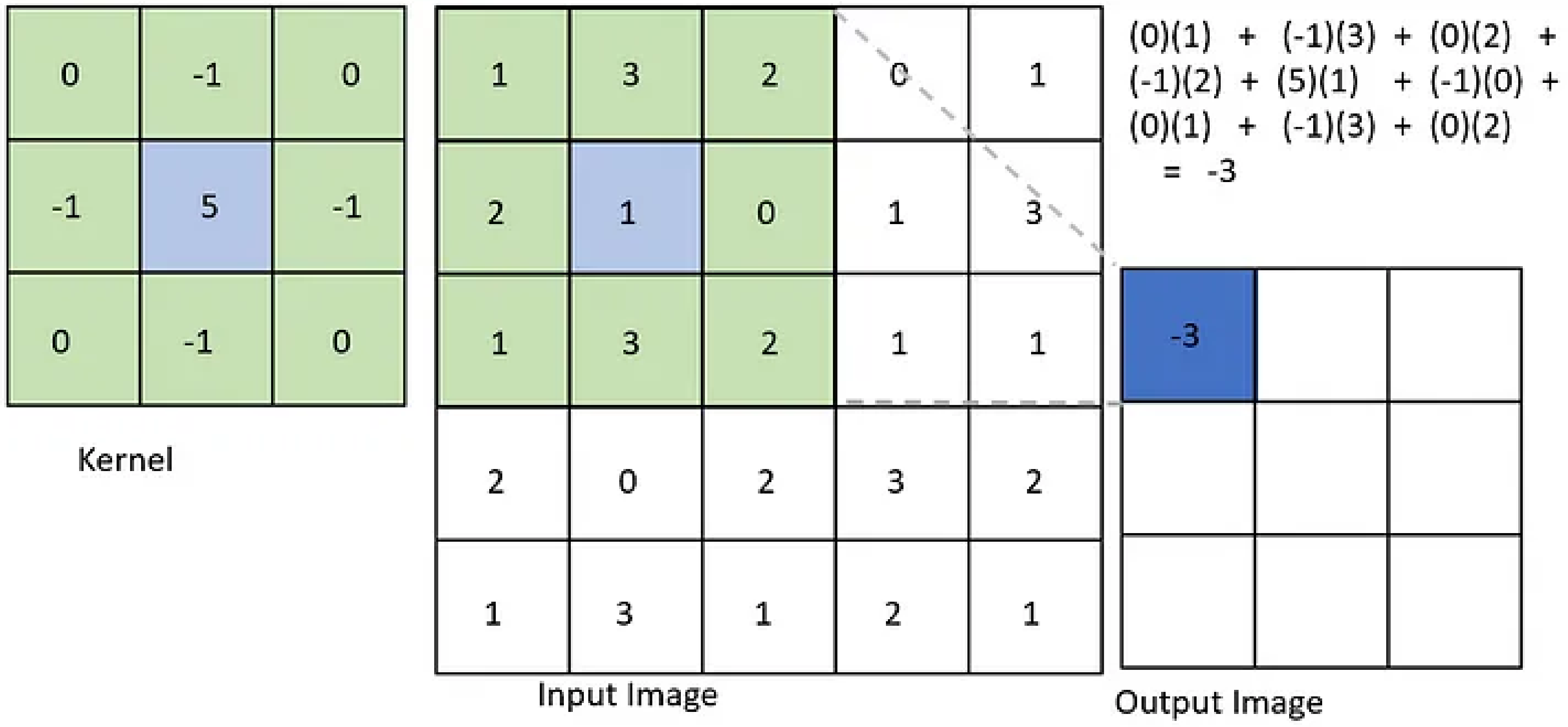
A filter (kernel) is systematically slid over a whole image.



Source: https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b

THE CONVOLUTION PROCESS

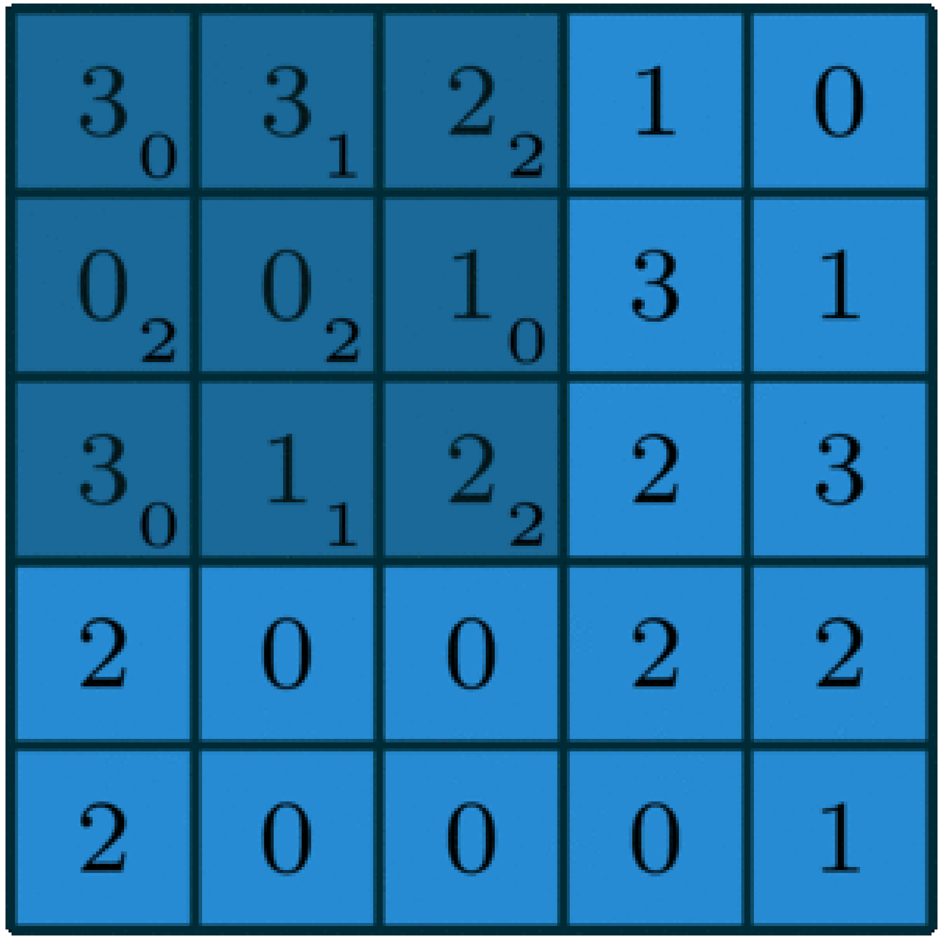
Pointwise multiplication takes place on each location, and the sum of them is assigned to the corresponding center pixel of the output image (feature map)..

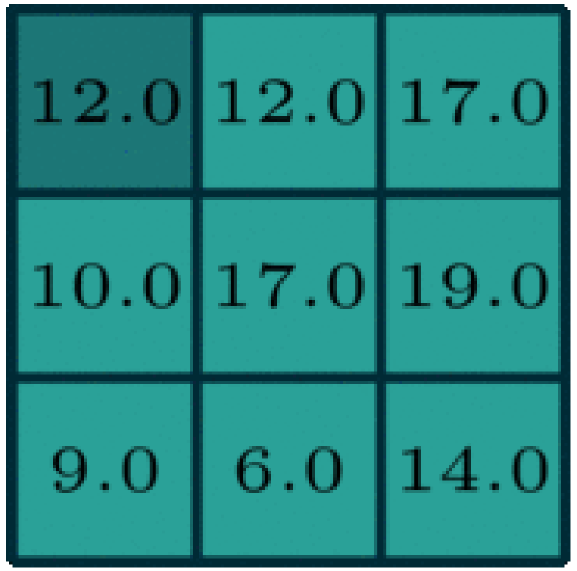


Source: https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b

THE CONVOLUTION PROCESS

The final output is a filtered image/feature map.

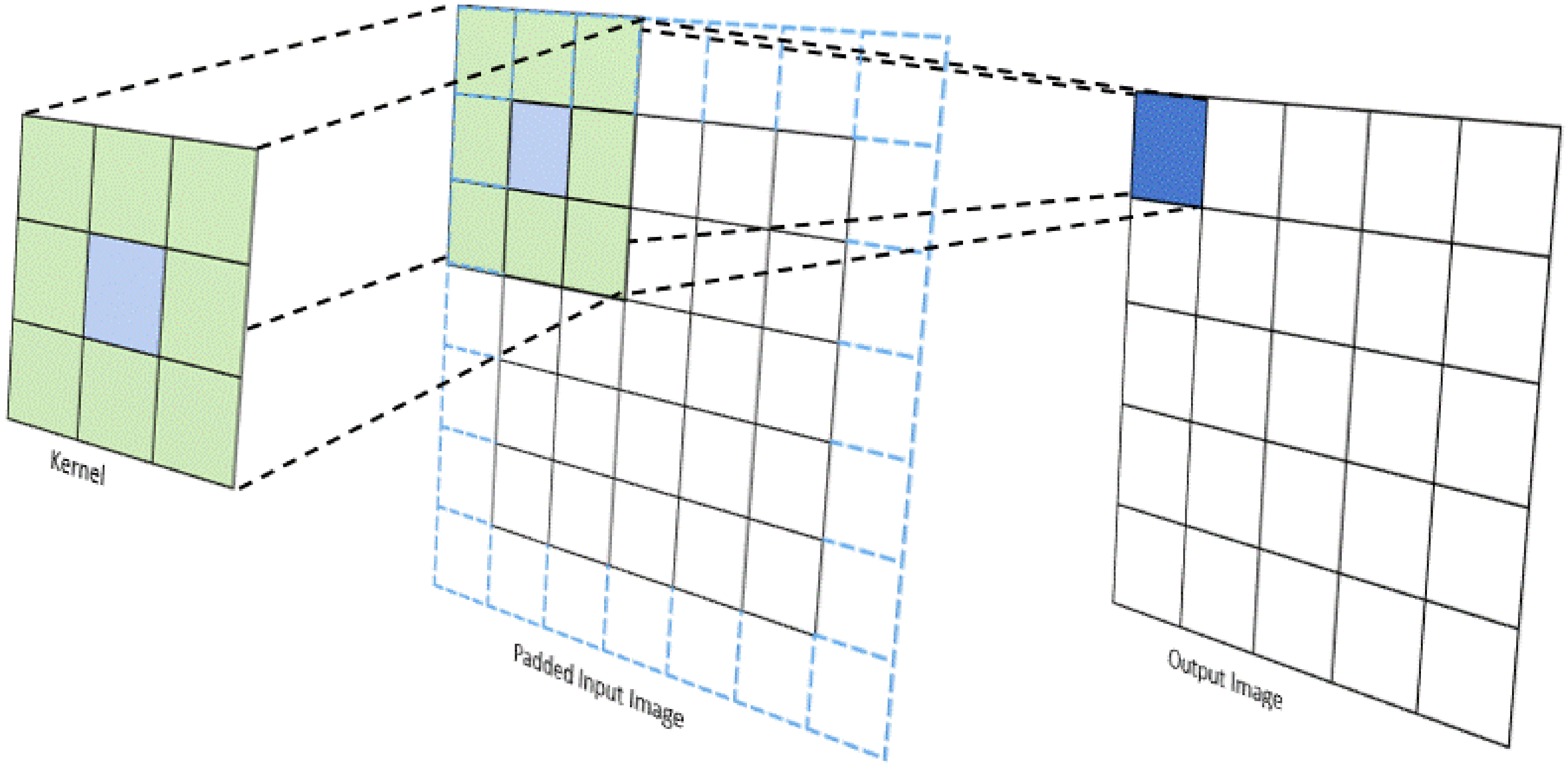




Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

THE CONVOLUTION PROCESS (PADDING)

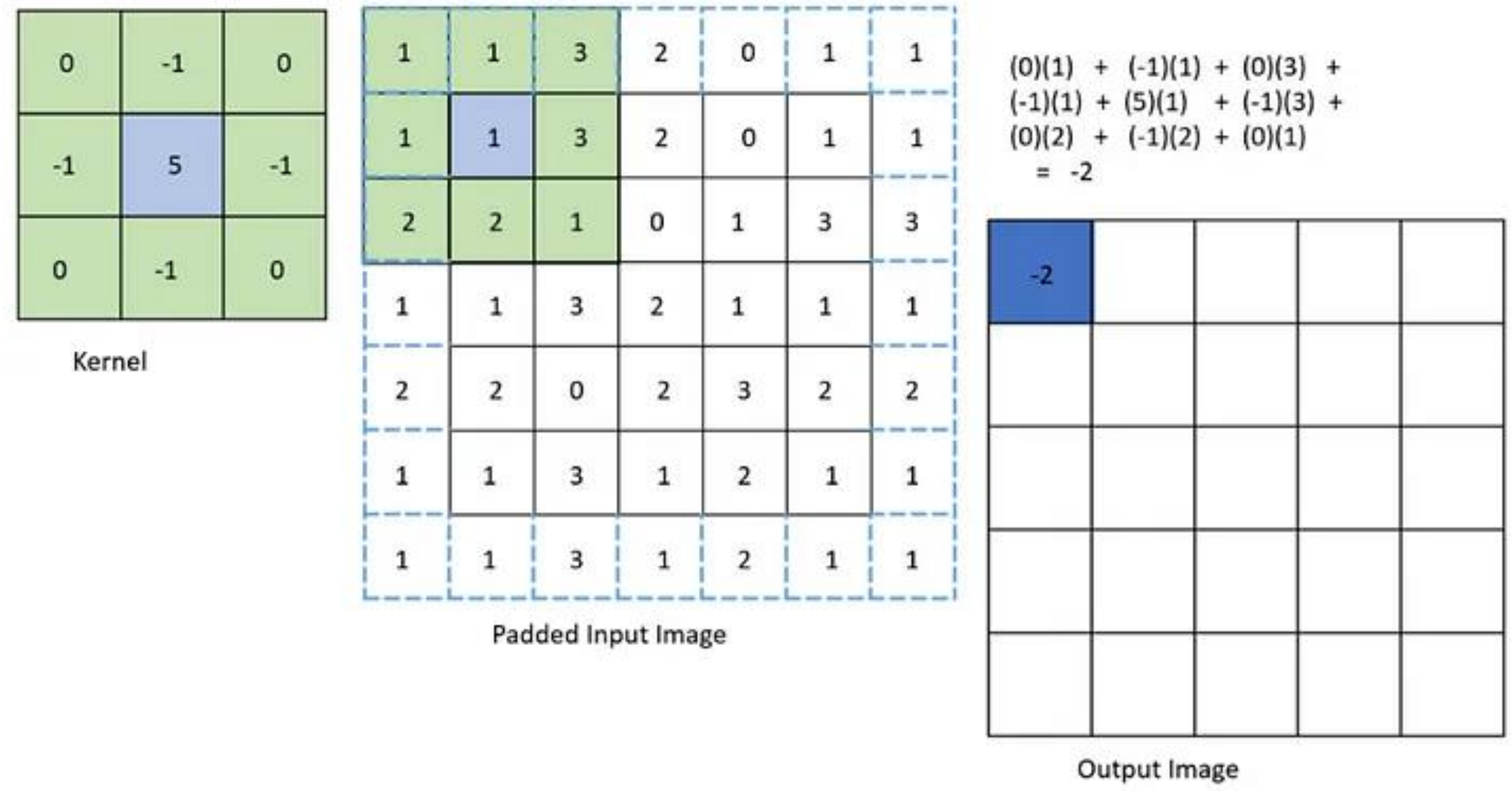
As we can see, Convolution may reduce the size of the image/signal. **Padding** extra layers of pixels around the borders can be done to preserve the original image's dimensions or to get any arbitrary shape. **Zero Padding, Edge Padding**.. These are some of the popular padding methods.



Source: https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b

THE CONVOLUTION PROCESS (PADDING)

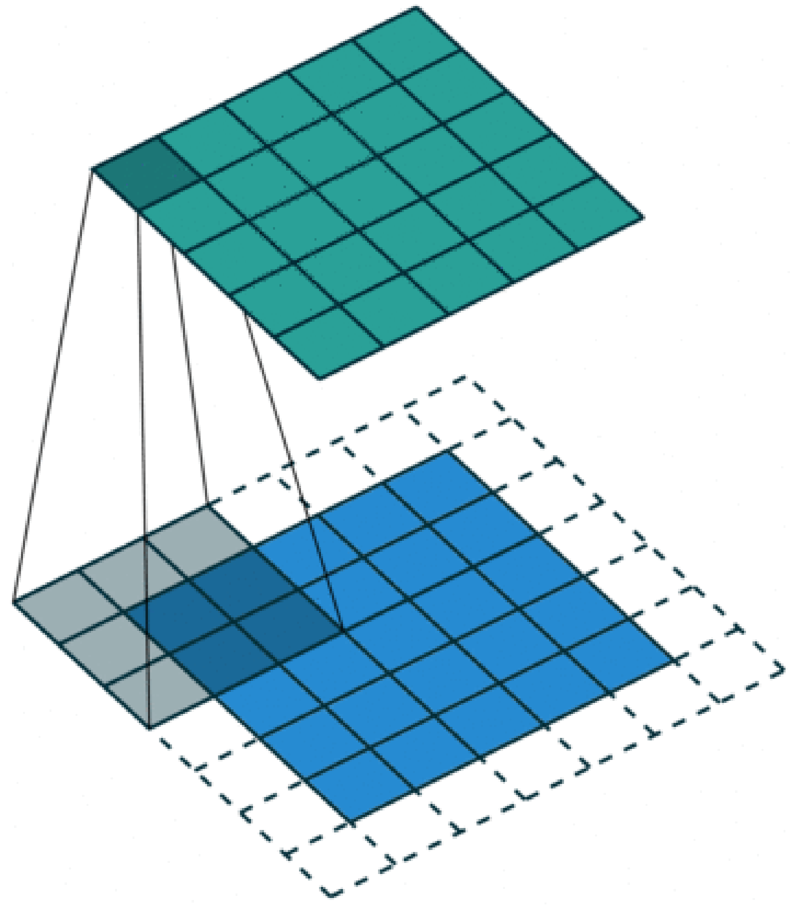
Padding 1 layer of pixels on each side preserves the dimension of the original image in the case of a 3×3 kernel.



Source: https://medium.com/@timothy_terati/image-convolution-filtering-a54dce7c786b

THE CONVOLUTION PROCESS (PADDING)

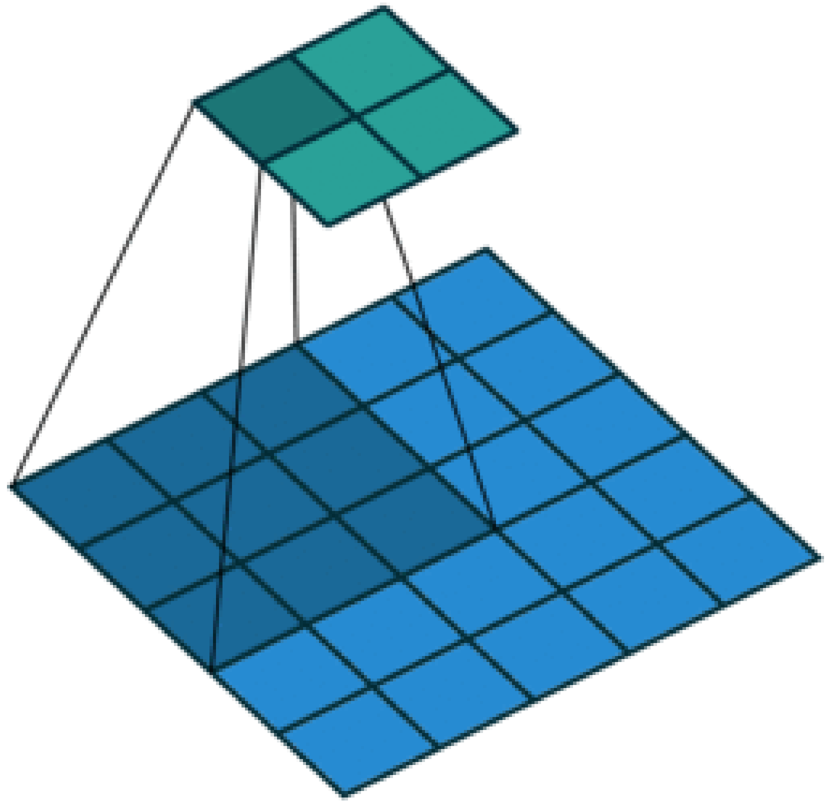
In which cases do we want to apply padding? Mainly **when the edges of the image contain useful information** that we want to capture.



Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

THE CONVOLUTION PROCESS (STRIDES)

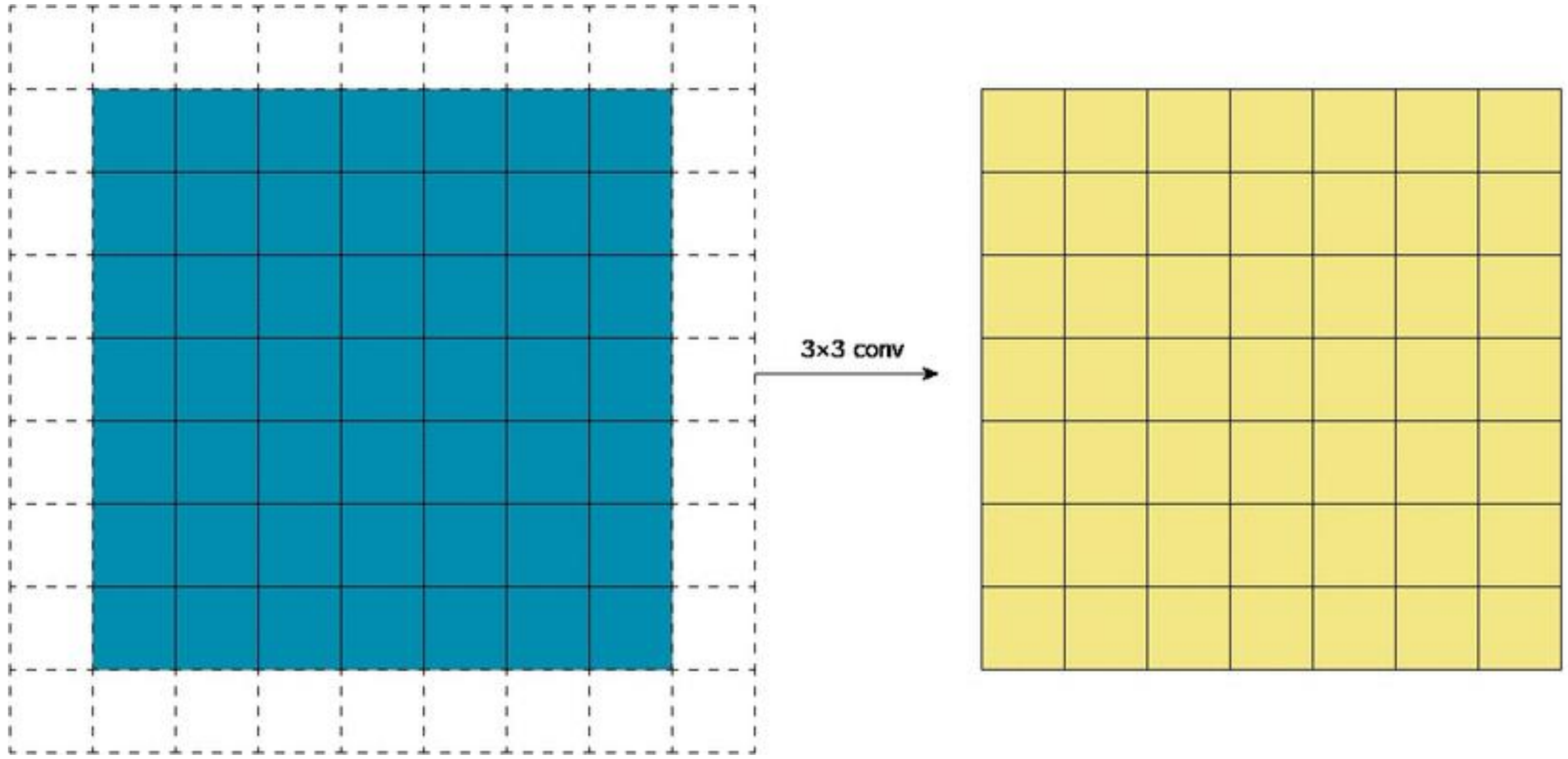
Stride defines how many pixels the filter (kernel) moves across the input image at each step. It controls how much we slide the filter horizontally and vertically during convolution. The animation below shows the convolution process with a **Stride = 2**.



Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

THE CONVOLUTION PROCESS (STRIDES)

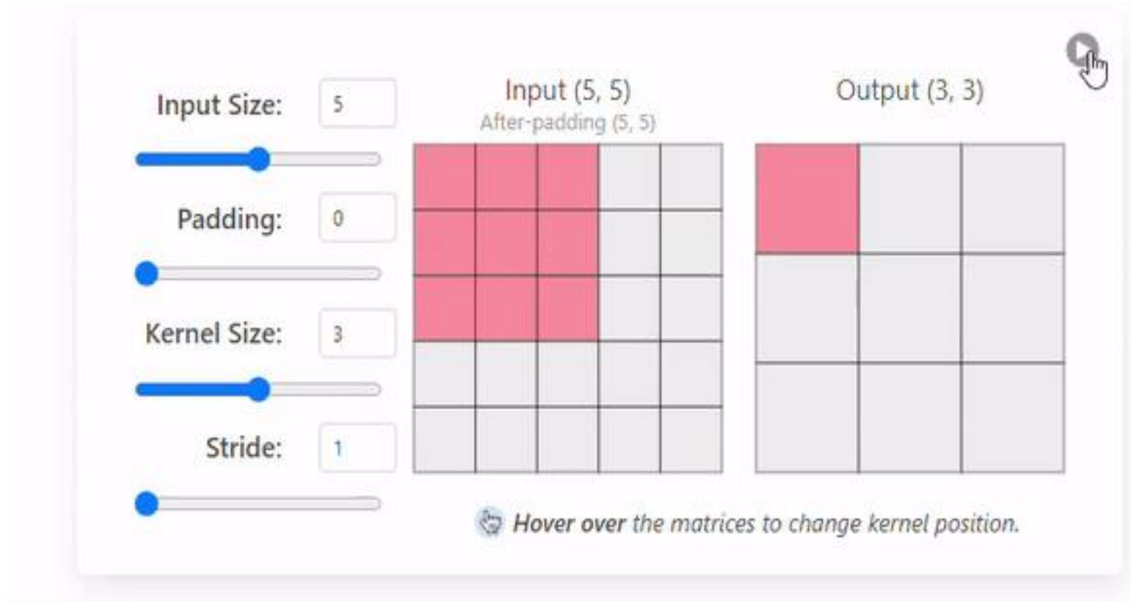
Clearly, increasing the number of strides results in smaller images/feature maps. This can be used to reduce spatial dimensions and thus, computation cost.



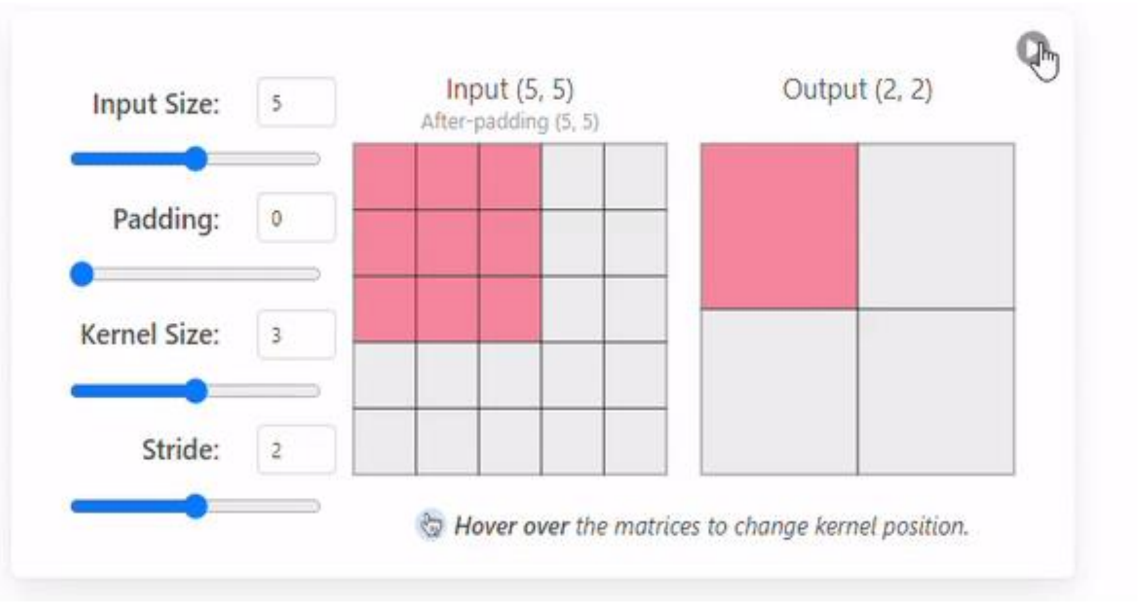
Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

THE CONVOLUTION PROCESS (STRIDES)

Observe the animations below to understand how different values of strides affect the dimensions of the output feature map.


Input Size: 5
Padding: 0
Kernel Size: 3
Stride: 1
Input (5, 5)
After-padding (5, 5)
Output (3, 3)
Hover over the matrices to change kernel position.

Stride = 1

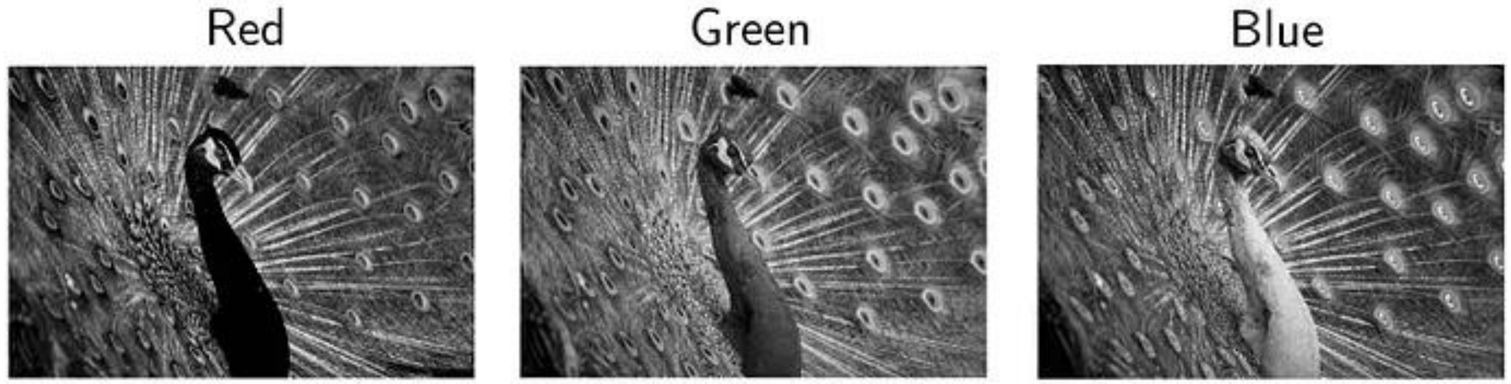

Input Size: 5
Padding: 0
Kernel Size: 3
Stride: 2
Input (5, 5)
After-padding (5, 5)
Output (2, 2)
Hover over the matrices to change kernel position.

Stride = 2

Source: https://poloclub.github.io/cnn-explainer/

CHANNELS

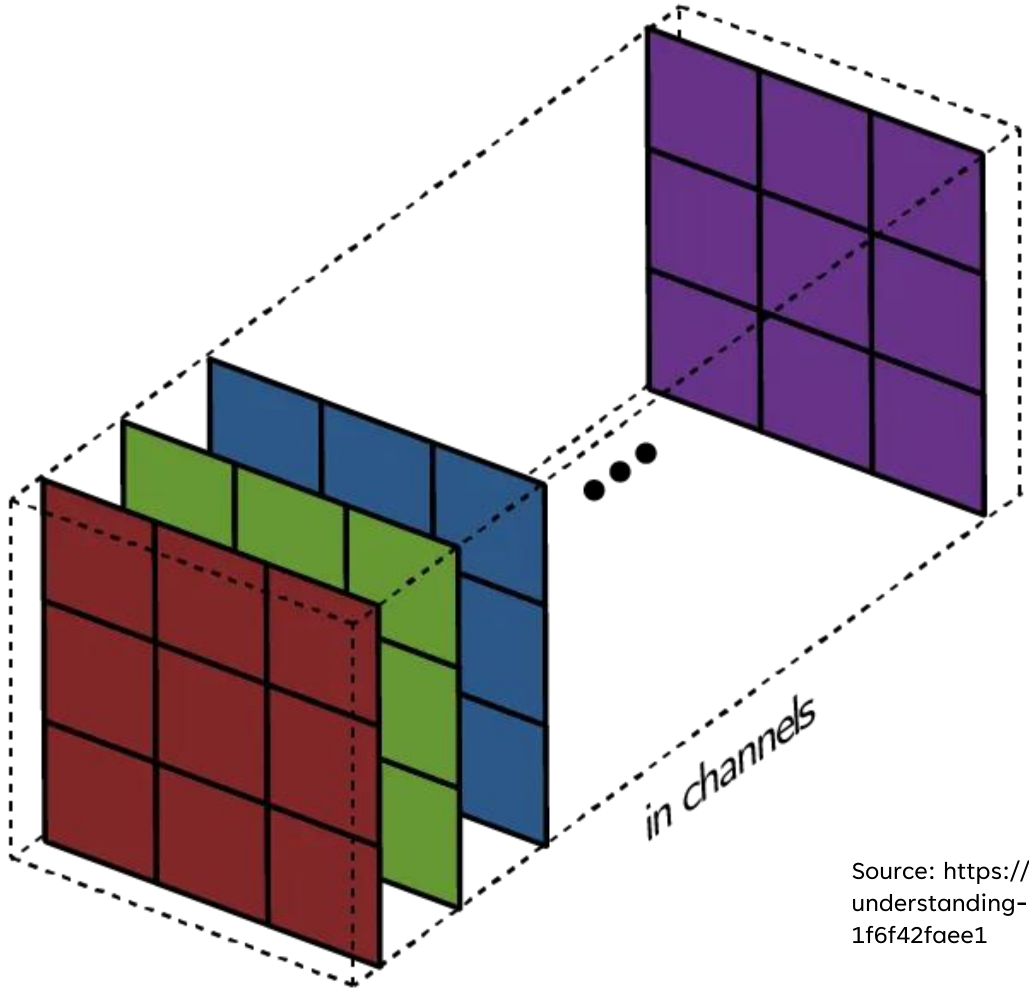
A **Channel** is like a separate layer of information in the image. A grayscale image has **1** channel (only **intensity**). An RGB image has **3** channels: **R (Red), G (Green),** and **B (Blue).** Some images (e.g., medical, hyperspectral) can have dozens of channels. We can think of channels like stacked sheets of paper: each sheet holds different information about the same position. For example, in RGB images, each channel contains information on different wavelengths of light (different colors).



Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

CHANNELS

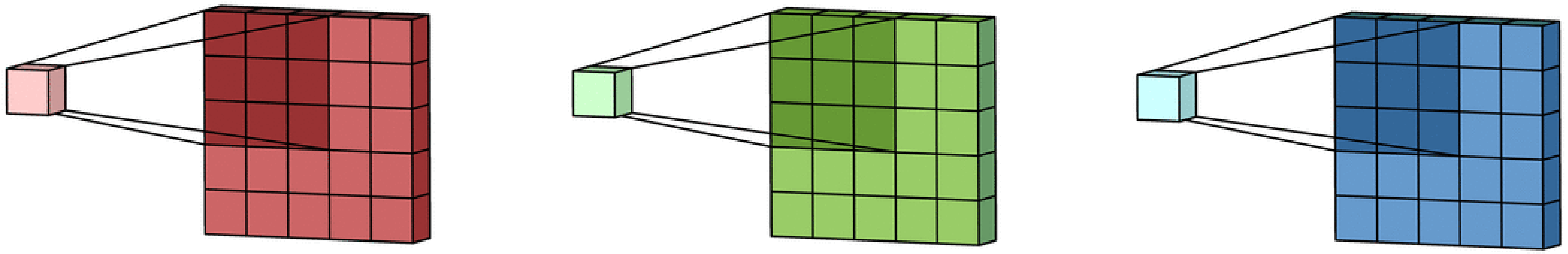
Instead of width × height (e.g., 224×224), an RGB image structure is **width × height × depth [channels]** (224×224×3). This is true for any image with multiple channels.



Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

CONVOLUTION ACROSS CHANNELS: KERNELS vs FILTERS

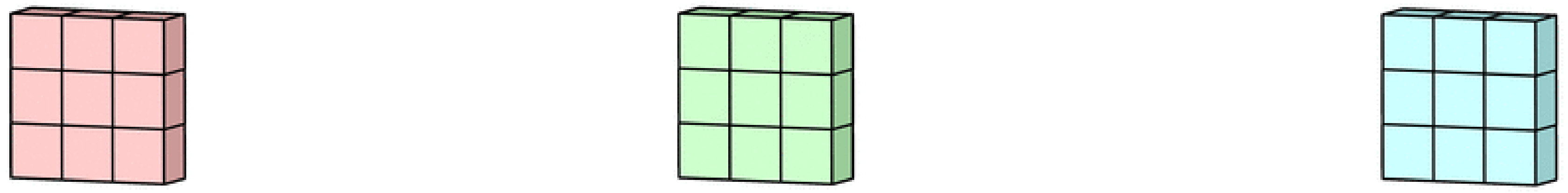
A single filter must process all channels together, not separately. **Each 3D filter will now have the same number of 2D kernels as the number of channels in the input image (One kernel for each channel).** So, an RGB image will need filters with 3 kernels (like channels in the image) each.



Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

CONVOLUTION ACROSS CHANNELS: KERNELS vs FILTERS

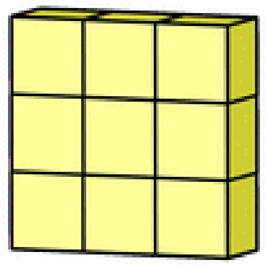
Each filter looks at an N×N patch (N = kernel dimension) across all 3 (or more) color channels at once. It produces one number for each spatial location, thus **1 output channel**. This one number is produced by **summing values from all filtered channels located in the same spatial location**.



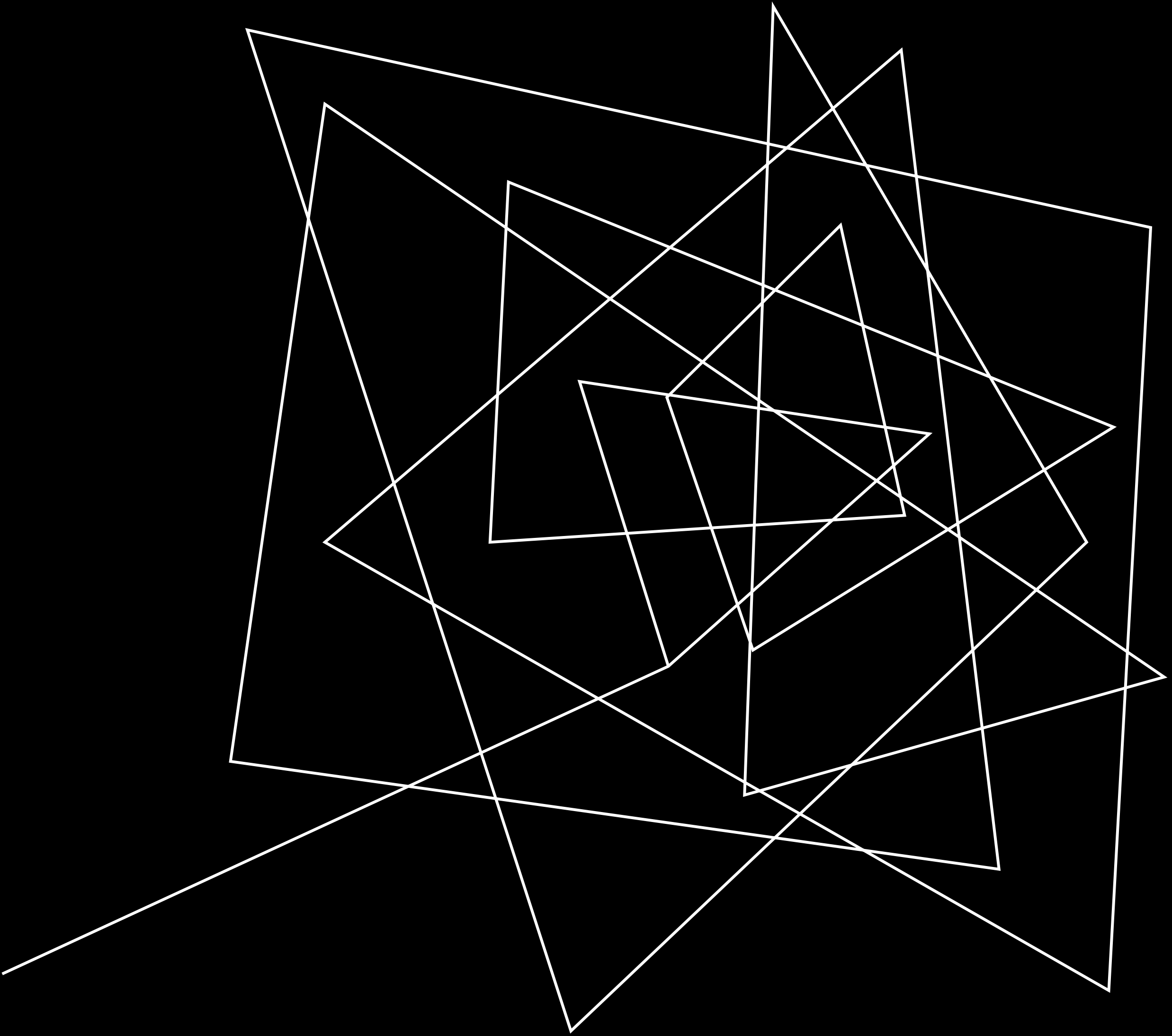
Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1

CONVOLUTION ACROSS CHANNELS

After summing, the CNN usually applies a bias and passes it through an activation function (like ReLU). The bias term allows the model to shift and adapt its activation, enabling the detection of features even when the input values are small or zero. Each filter has one bias term.



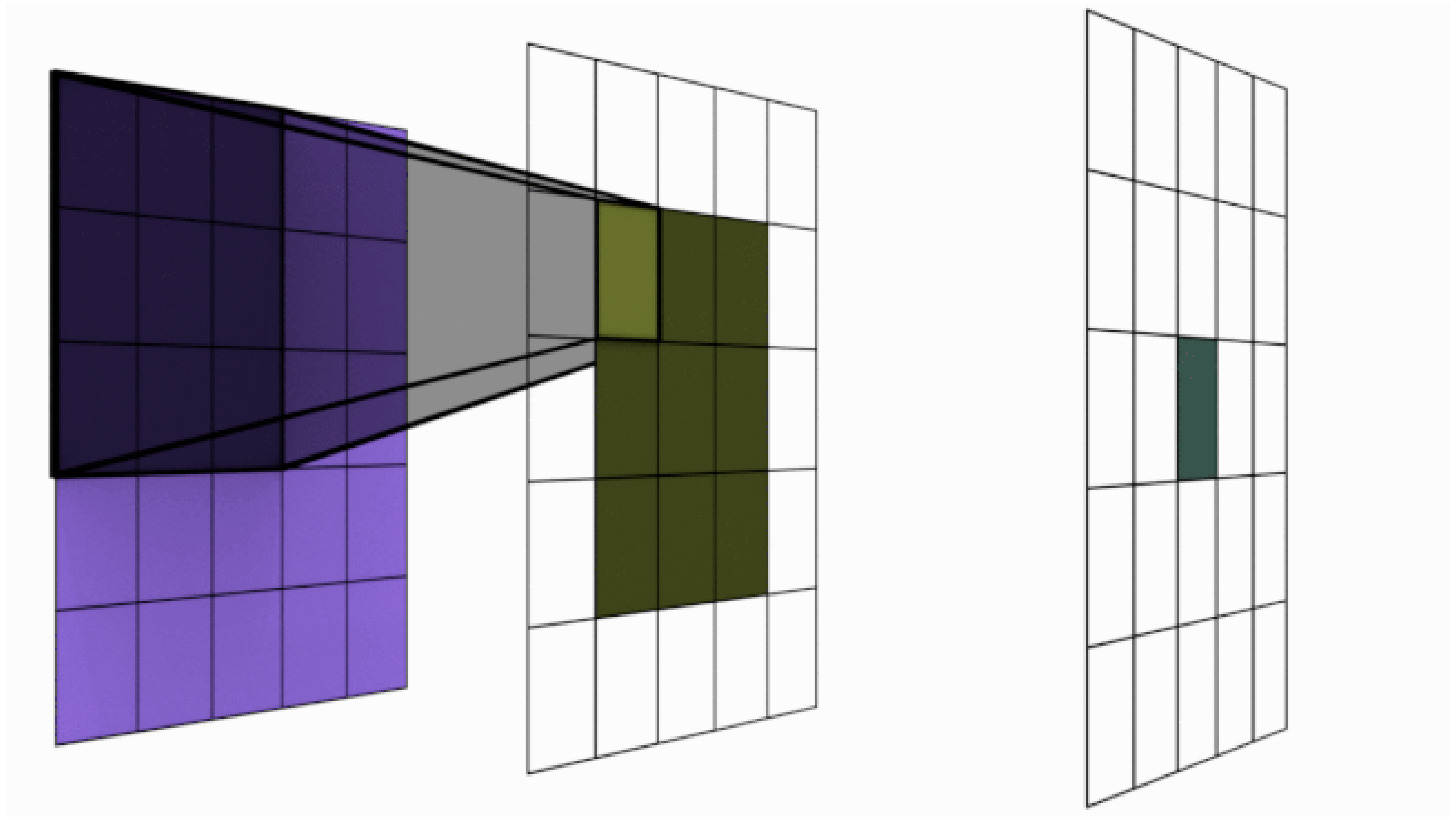
Source: https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1



RECEPTIVE FIELDS

RECEPTIVE FIELDS

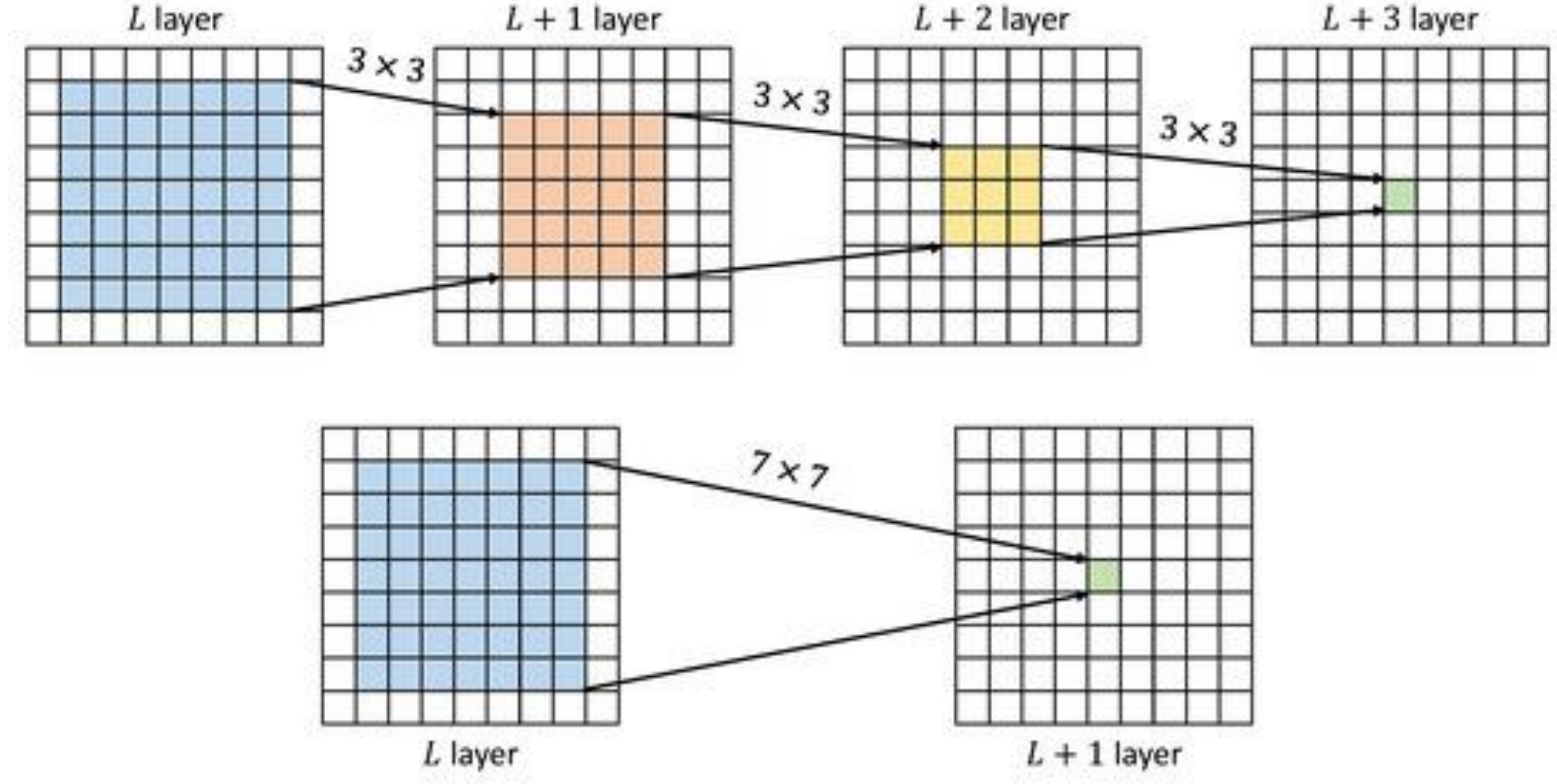
The **Receptive Field** of a neuron in a CNN is **the region of the input image that affects the value of that particular neuron**. For example, in the first layer, receptive field = size of the filter (e.g., 3×3 for filters of dimension 3). In deeper layers, the receptive field grows larger because **each neuron indirectly depends on a larger region of the input** (**“Seeing" bigger contexts** of the input).



Source: https://www.linkedin.com/pulse/deep-learning-understanding-receptive-field-computer-n-bhatt

RECEPTIVE FIELDS

The image below shows how receptive fields become larger **in the deeper layers,** as well as **when the size of the filter is larger**.



Source: https://doi.org/10.3390/app13063403

WHY RECEPTIVE FIELDS ARE IMPORTANT

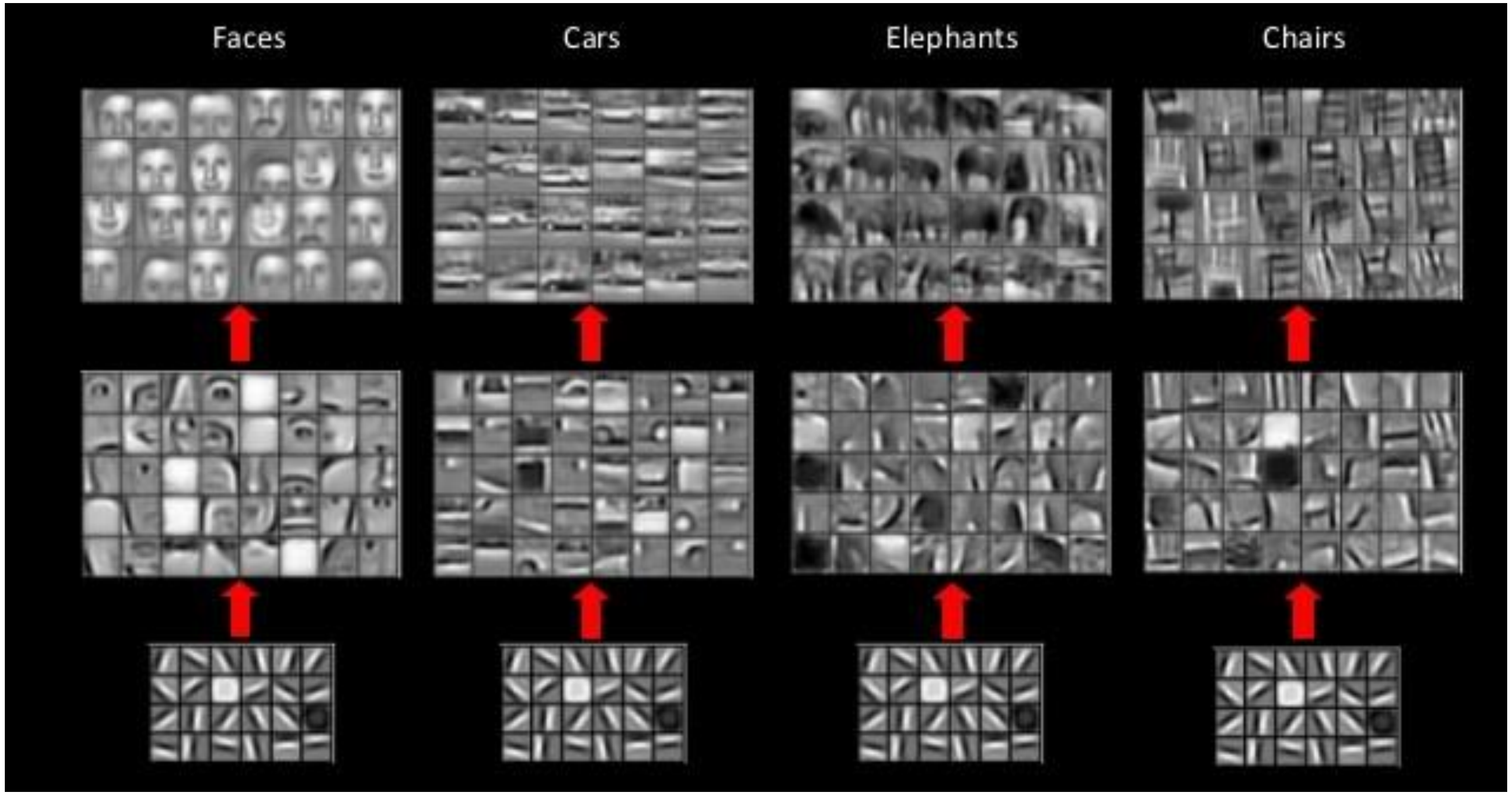
Receptive field measures how much of the original input a neuron 'sees'. Deeper neurons have larger receptive fields, enabling detection of complex and large-scale patterns. This process can be compared to the foveal and peripheral vision of the human visual system.

- **Small Receptive Fields act like Foveal Vision**. When we read a word, our eyes focus on just a few letters. We're picking up **fine details**. This is like the early CNN layers with small receptive fields. These detect **edges, textures, colors** — very **local features**.
- **Large Receptive Fields play the role of Peripheral Vision.** As we look around a scene, our peripheral vision helps us understand the **overall layout** — where the objects are, who's standing where. This is like deeper CNN layers. They integrate signals from many small details to build up **semantic understanding**. They detect **faces, animals, traffic signs**, etc.

In short, CNNs work like our visual system. We first see local details (edges, corners) using smaller receptive fields and gradually build a global picture (faces, objects) using larger receptive fields. Also, this process enables **detecting hierarchical features as we go deeper into the CNN**. From lines/edges (earlier layers) → shapes → objects (deeper layers), like from letters → words → sentences. That’s why convolutional layers are known as **feature extractors**.

WHY RECEPTIVE FIELDS ARE IMPORTANT

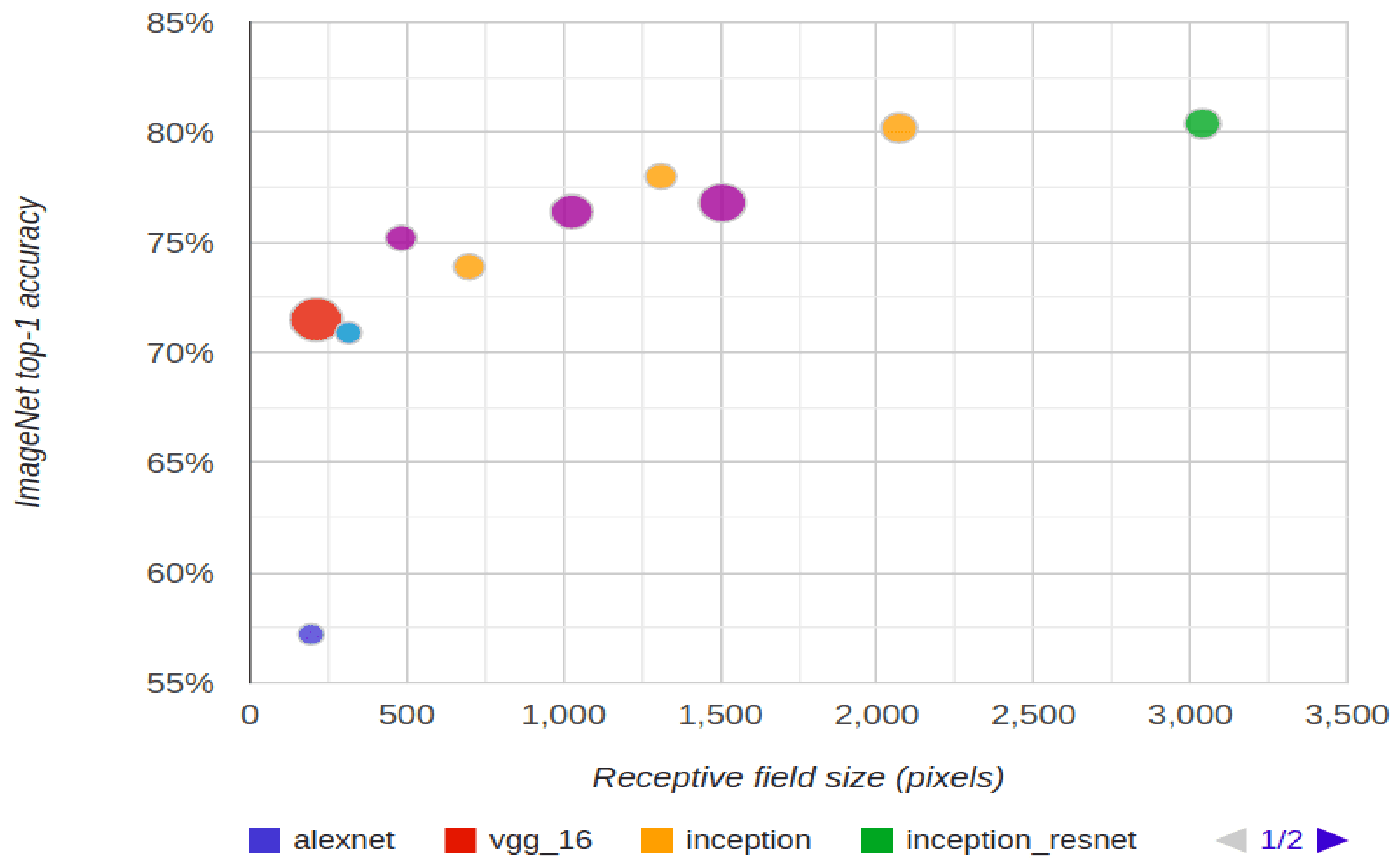
The image below shows how CNN layers capture lower-level features like lines/edges (local patterns) in the earlier layers and gradually move towards capturing more complex features like eyes, faces (global shapes/objects) as we go deeper into the network.



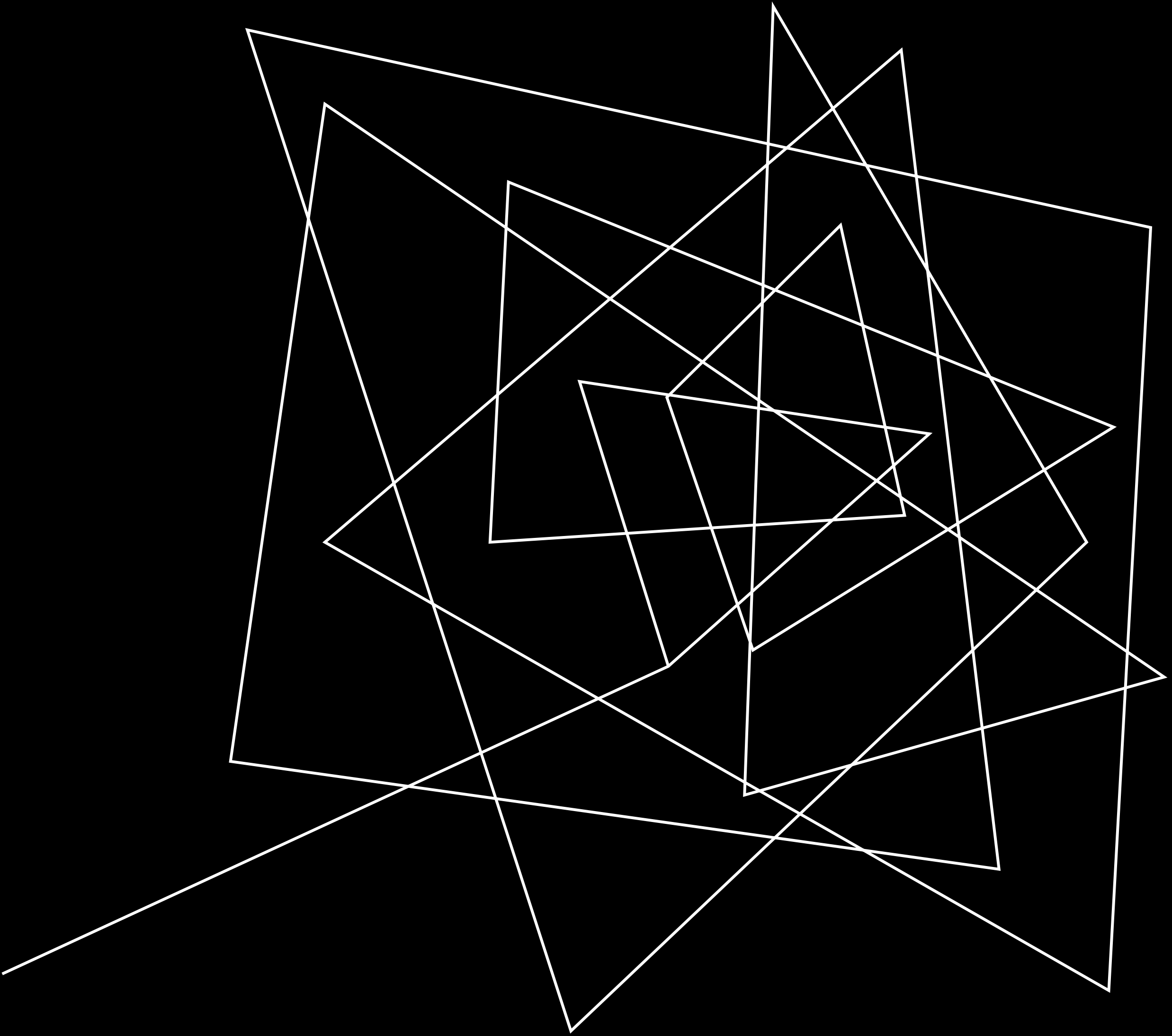
Source: https://www.linkedin.com/pulse/deep-learning-understanding-receptive-field-computer-n-bhatt

WHY RECEPTIVE FIELDS ARE IMPORTANT

In the graphic below, we observe a logarithmic relationship between classification accuracy and receptive field size, which suggests that **large receptive fields are necessary for high-level recognition tasks, but with diminishing returns**.



Source: https://theaisummer.com/receptive-field/



LAYERS OF CNN & CNN ARCHITECTURE DESIGN

LAYERS OF CNN: THE BUILDING BLOCKS

An end-to-end **Convolutional Neural Network (CNN)** typically contains several types of layers, each with different mechanism and purpose. Each layer transforms the input and feeds richer representations forward. Together, these layers form the powerful CNN architecture which can perform various ML tasks.

Common Layers of a CNN

- Convolutional Layer
- Pooling Layer
- Flattening Layer
- Fully Connected (Dense) Layer
- Activation Layer
- Batch Normalization Layer
- Dropout Layer

Some of these layers (like activation or dropout layers) are sometimes not explicitly mentioned and considered part of other layers. But roughly, these are the most common layers used in a CNN.

CONVOLUTIONAL LAYER

Convolutional layers are the fundamental building blocks of CNNs. These layers perform a critical mathematical operation known as convolution (As we learned already). This process entails the application of specialized filters known as kernels, that traverse through the input image to learn complex visual patterns.

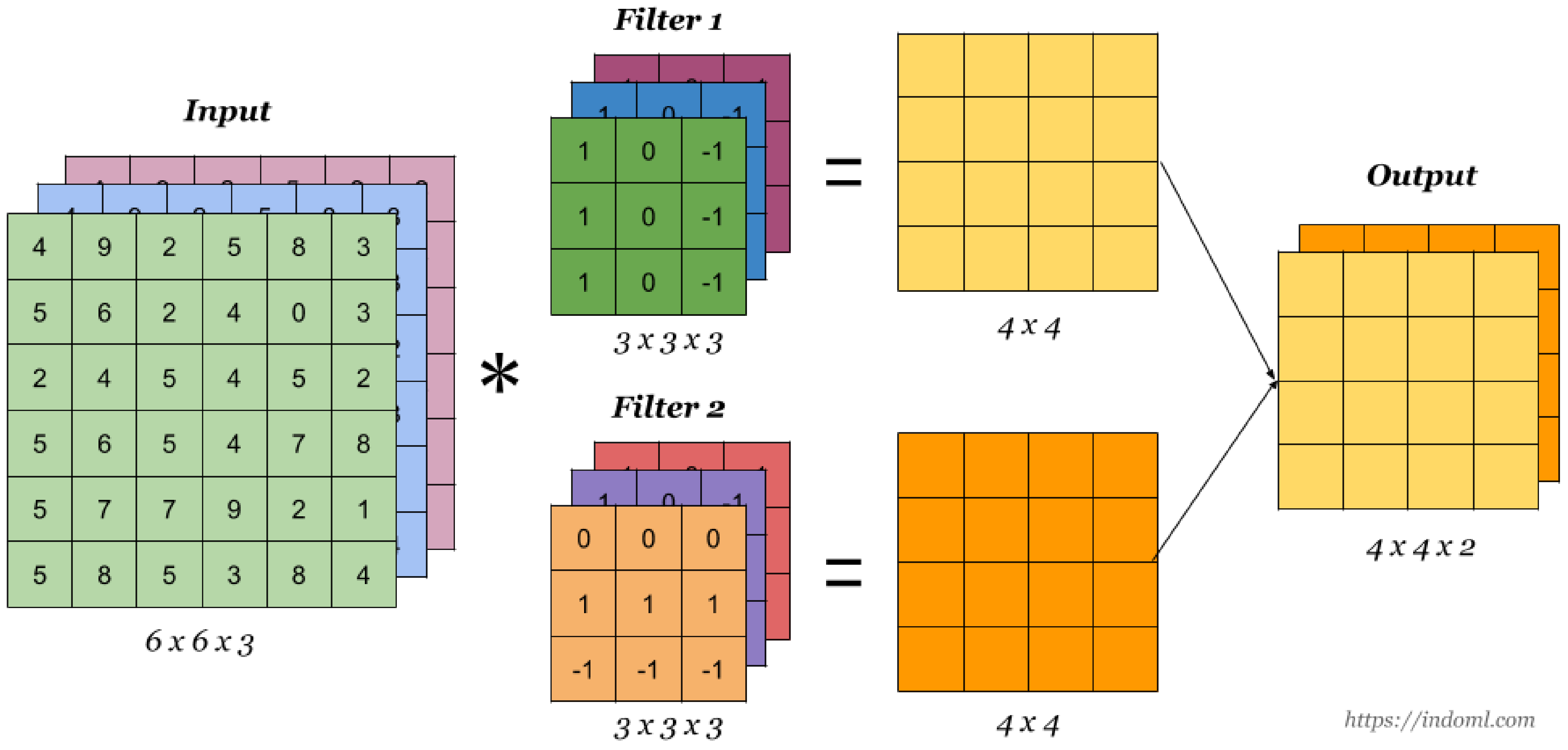
Purpose: Applies not one but **a set of filters** (learned kernels) over the input, producing multiple feature maps as output. These feature maps help to:

- **Detect local patterns** within the receptive field (e.g., edges, textures, shapes).
- **Learns what to look for** (e.g., vertical edge, circle, eye, etc.).

How exactly does it become so useful? How can we extract image features by just multiplying and adding pixels? We have already covered the concept of receptive fields. But to understand more clearly, you can watch this public YouTube video: https://youtu.be/KuXjwB4LzSA?si=DOv5U_bzgZPGgzVO .

CONVOLUTIONAL LAYER

The image below underlines how multiple filters help to extract different features. By applying **multiple filters**, a convolutional layer creates **a set of feature maps, each representing a different aspect of the input data.**



POOLING LAYER

Although Convolutional Layers can decrease the output size, **their principal objective is not Dimensionality Reduction**. The main objective of Convolutional Layers is **Feature Extraction**.

Then there are **Pooling Layers**, and their main objective is indeed **dimensionality reduction.** Suppose we have a large image and want to make it smaller but keep all the important features like edges and colors. The pooling layer operates independently on every depth slice of the input. It resizes it spatially, using the Max or Average of the values in a window slid over the input data.

Pooling Layers reduce spatial dimensions (**not channels**), while **retaining dominant features.** We can think of pooling as **summarizing regions of feature maps**. *In pooling, we are not applying any kernel to the input data, we are just simplifying the information with a math operation (Max or Avg or something similar).* It helps to:

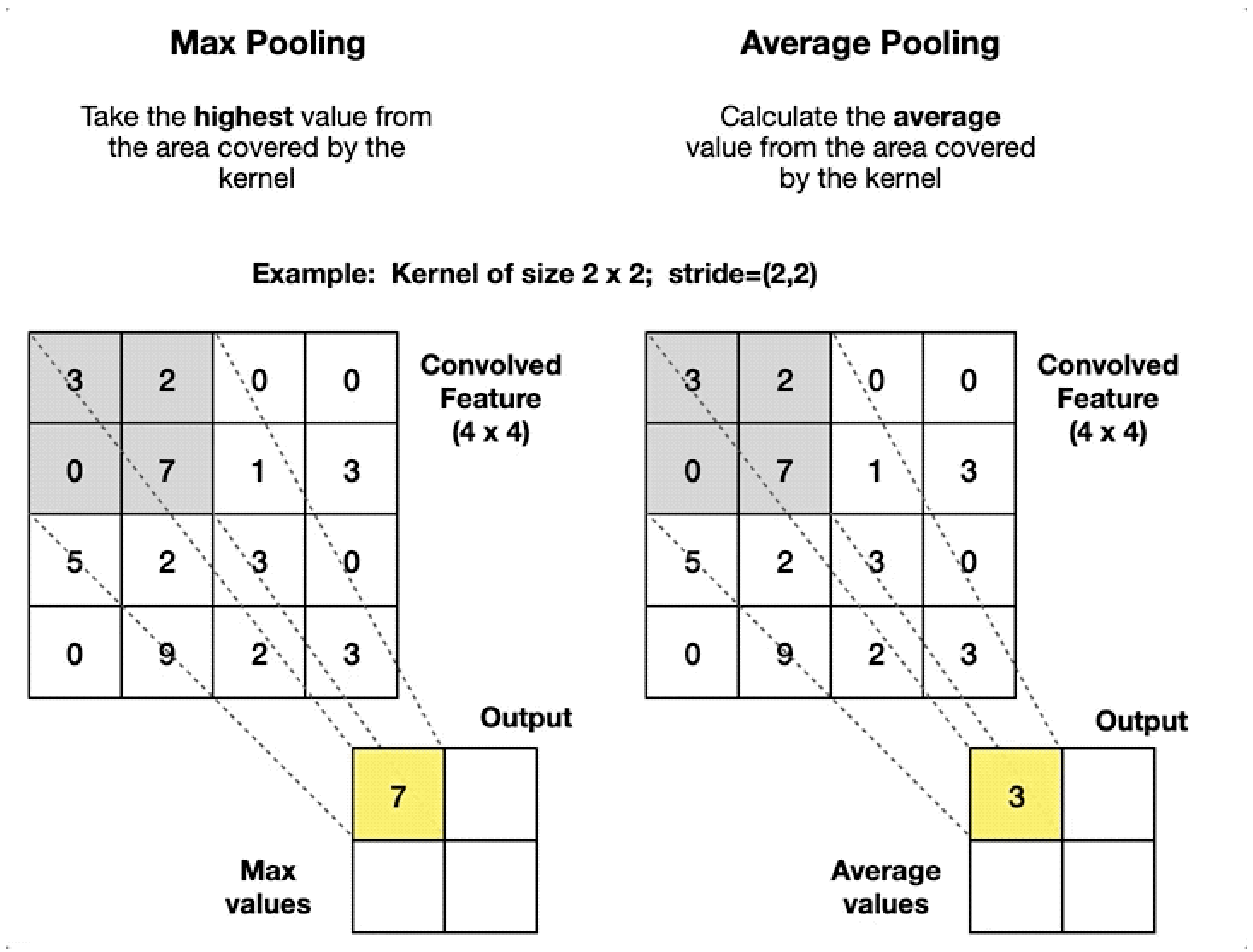
- Reduces computation and memory.
- Adds translational invariance.
- Helps generalization.

POOLING LAYER

The purpose of Pooling is to **gradually shrink the representation's spatial dimension to minimize the number of parameters and computations in the network**. There are several types of Pooling. The most common types are:

- **Max Pooling**: This works by selecting the maximum value from every pool. Max Pooling retains the most prominent features of the feature map, and the returned image is sharper than the original image.
- **Average Pooling**: This pooling layer works by getting the average of the pool. Average pooling retains the average values of features of the feature map. It smoothens the image while keeping the essence of the feature in an image.
- **Global Pooling Layers:** Global Pooling Layers, particularly **Global Average Pooling** often replace the classifier's fully connected or Flattening layer (to be discussed). The model instead **ends with a convolutional layer that produces as many feature maps as there are target classes** and **performs global average pooling on each of the feature maps to combine each feature map into a single value**.

POOLING LAYER



Max Pooling

Take the **highest** value from the area covered by the kernel

Average Pooling

Calculate the **average** value from the area covered by the kernel

Example: Kernel of size 2 x 2; stride=(2,2)

Source: https://towardsdatascience.com/convolutional-neural-networks-explained-how-to-successfully-classify-images-in-python-df829d4ba761/

FLATTENING

After convolutional and pooling layers have extracted relevant features from the input image, we have to turn this high-dimensional feature map into a format suitable for feeding into fully connected layers (which acts as the classifier/decision maker).

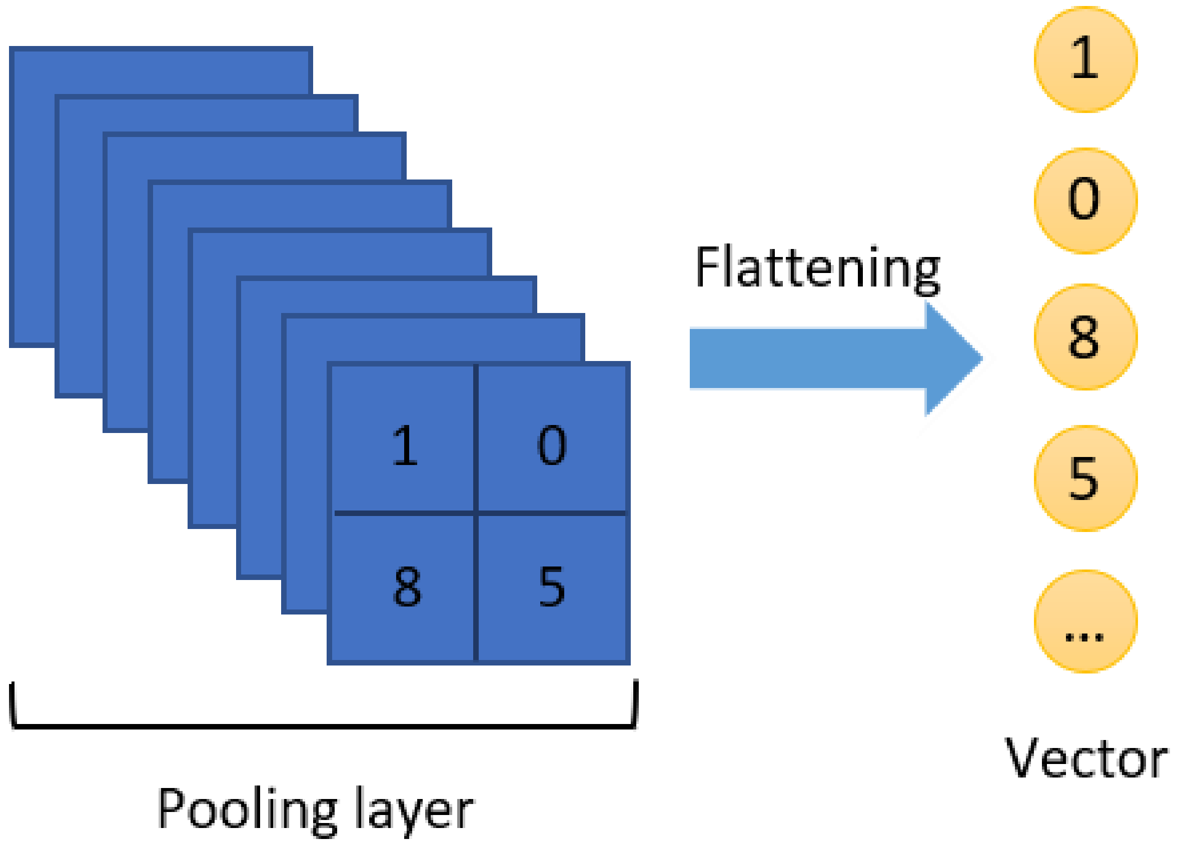
At this point, we have a grid of data (pixels in a feature map), and we want to line up all of these grid points in a single, long line. That's what **Flattening** does. It **takes the entire feature map** and **reorganizes it into a single, long vector**.

Why do we need flattening layers?

- **Integration of features:** By flattening the feature maps into a vector, the network can integrate the spatially distributed features extracted for tasks such as classification.
- **Compatibility with Dense Layers:** Fully connected layers (dense layers) are designed to operate on 1-dimensional data. Hence, flattening is a necessary step to transition from the multidimensional tensors produced by convolutional layers to the format required for dense layers.

FLATTENING

Although flattening changes the shape of the data, it does not make any changes to the actual information.



Source: https://blog.gopenai.com/understanding-autoencoders-part-iv-the-fully-connected-layer-and-flattening-566f246fa1a

FULLY CONNECTED/DENSE LAYERS

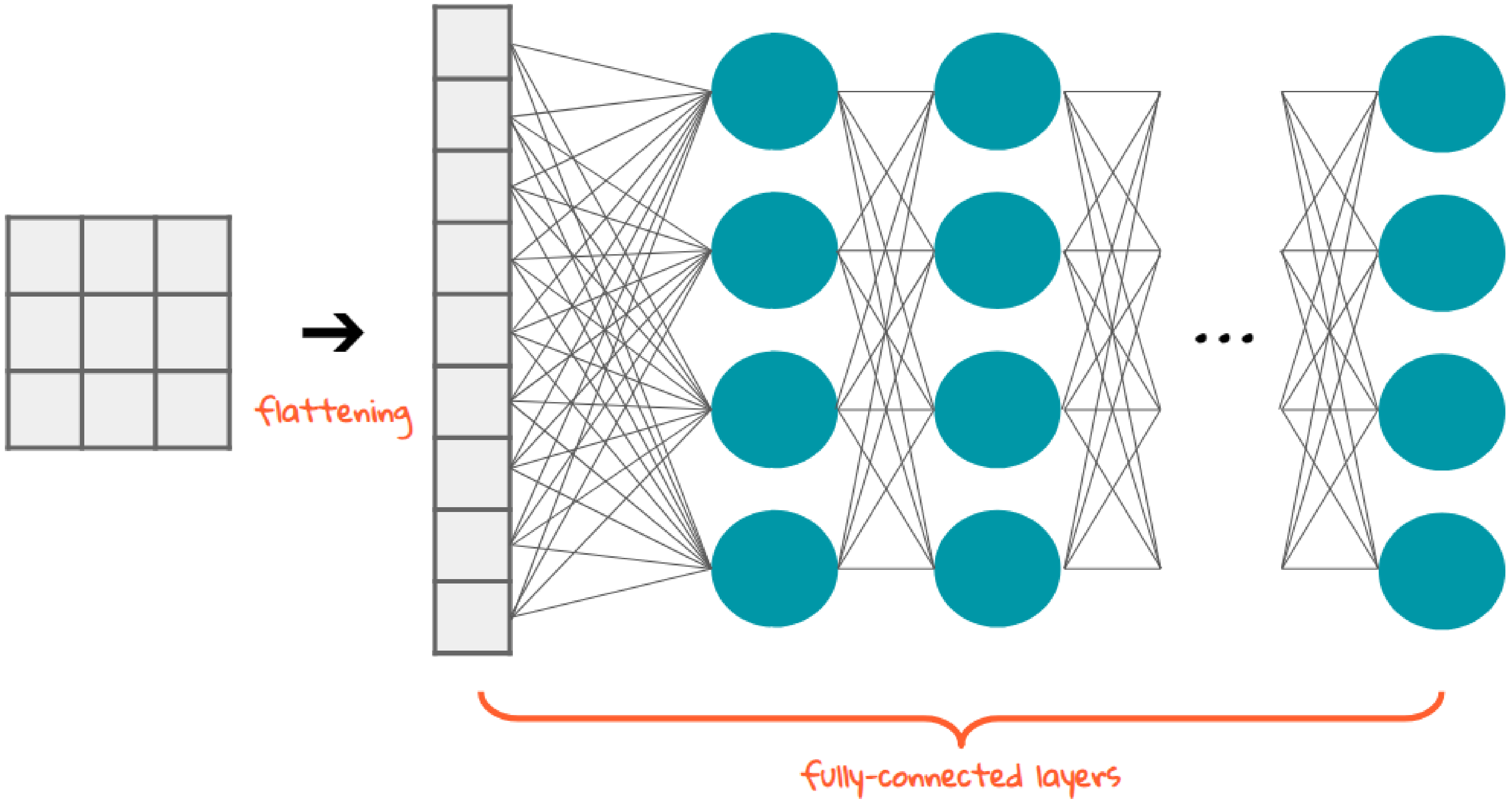
Dense layers take input from the flattening layer and the rest of the structure is like the *feedforward neural networks* we saw before. While convolutional layers are good at detecting features in input data, **Dense layers are essential for integrating these features into predictions**. For example, if we design a convolutional neural network for facial recognition, early convolutional layers might detect edges and textures, while dense layers might interpret these to recognize specific facial features. So, the dense layers act as the **decision-maker** based on the **features extracted earlier**.

Purpose

- Dense layers turn abstract features into concrete predictions. They are the **decision-makers** after CNN layers do the seeing.
- Dense layers connect every neuron from the previous layer, allowing them to **combine features** from different regions of the image. They can also **integrate global spatial information** that pooling and convolution may miss.
- **Output Layer:** Dense layers are also typically used at the end of the network as output layers. For classification, they output class scores through some activation (Sigmoid or SoftMax). For regression they output a continuous value.

FULLY CONNECTED/DENSE LAYERS

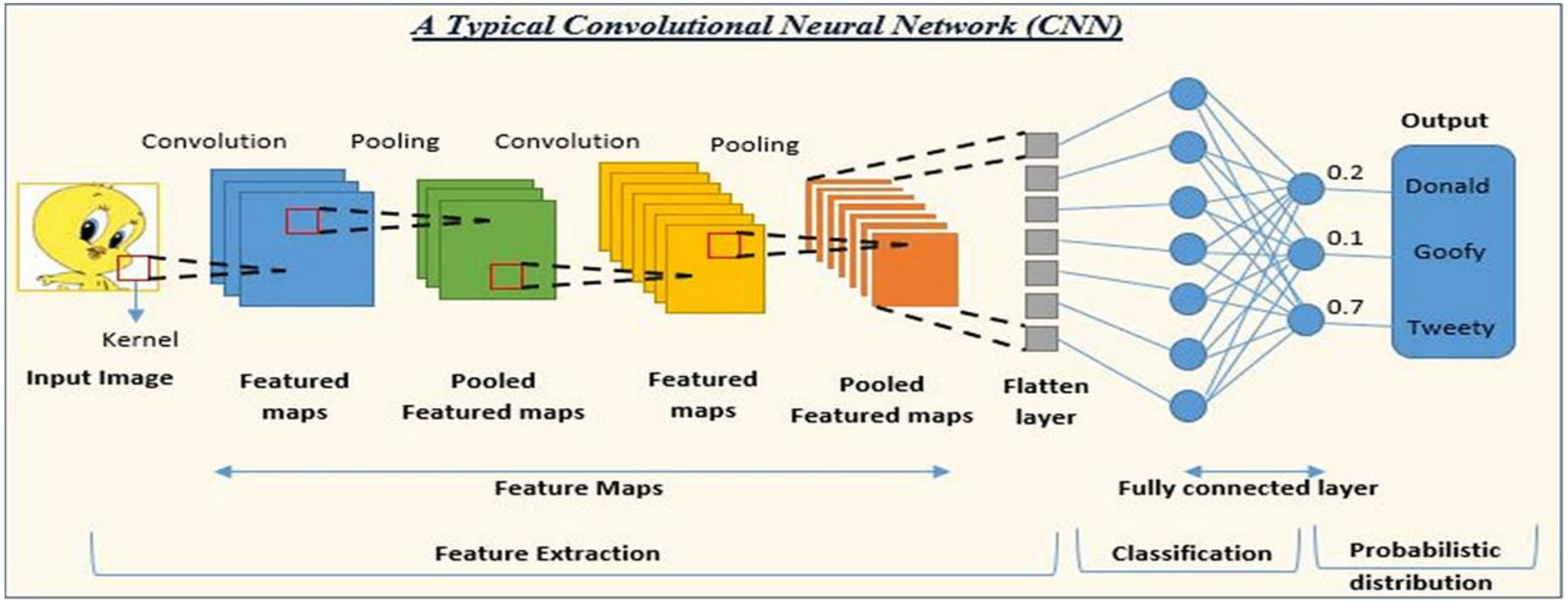
Typically, **at least one** Dense Layer is needed at the end of a CNN architecture. However, the number of dense layers and their neurons can vary widely.



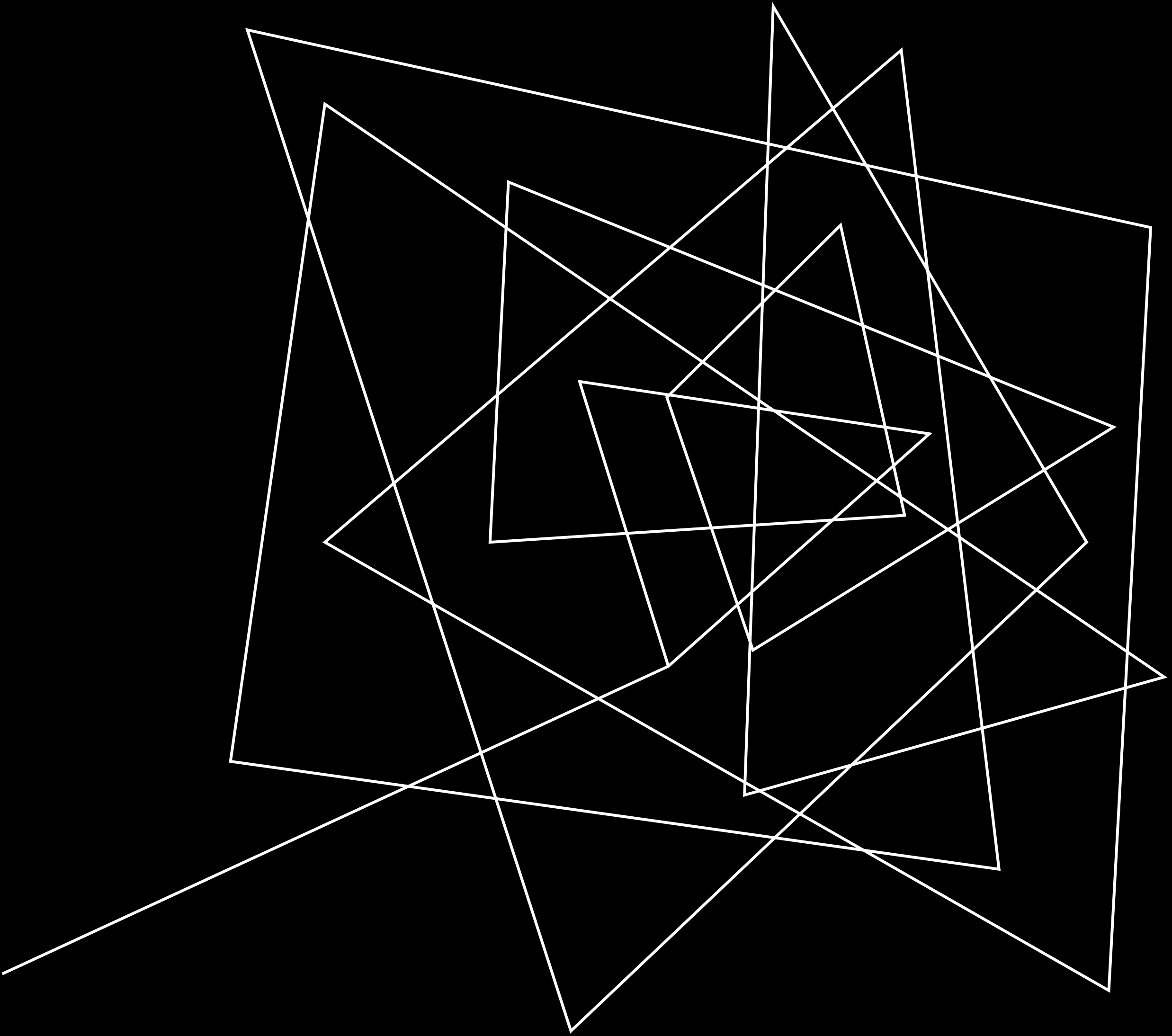
Source: https://mohamedbakrey094.medium.com/a-simplified-explanation-of-cnn-component-layers-in-deep-learning-518b5c45ec8c

A TYPICAL CNN STRUCTURE

We can put all these layers together to form a whole CNN structure. Typically, a CNN starts with some **Convolutional + Pooling layers** (Conv. layers followed by Pooling) → **Flattening** → Followed by a few **Dense layers** → **Output layer.** Activations are used after each Conv./Dense layer, and Batch Normalization and Dropout layers are often added in between to enhance performance and generalization.



Source: https://python.plainenglish.io/understanding-cnns-and-resnets-a-beginners-guide-with-python-code-f4afb19cf4e4



INPUT/OUTPUT DIMENSIONS & PARAMETER CALCULATION

INPUT/OUTPUT DIMENSIONS

1. Convolutional Layer

Let, Input Dimensions → **[W_{in}, H_{in}, C_{in}]**

Output Dimensions → **[W_{out}, H_{out}, C_{out}]**

Then,

$$W_{out} = \frac{W_{in} - F + 2*P}{\#Stride} + 1$$

$$H_{out} = \frac{H_{in} - F + 2*P}{\#Stride} + 1$$

C_{out} **= Number of Filters applied on the input images**

Where, **F** = Kernel Dimension (**F** for an **F*F** kernel)

P = Padding Length

#Stride = Strides (Default stride is **1** for Conv layers)

INPUT/OUTPUT DIMENSIONS

1. Convolutional Layer (For Batch Size N)

Let, Input Dimensions → **[N, W_{in}, H_{in}, C_{in}]**

→ Output Dimensions → **[N, W_{out}, H_{out}, C_{out}]**

Then,

$$\boxed{\begin{array}{l} W_{out} = \dfrac{W_{in} - F + 2*P}{\#Stride} + 1 \\[2em] H_{out} = \dfrac{H_{in} - F + 2*P}{\#Stride} + 1 \\[2em] C_{out} = \text{Number of Filters applied on the input images} \end{array}}$$

Where, **F** = Kernel Dimension (**F** for an **F*F** kernel)
P = Padding Length
#Stride = Strides (Default stride is **1** for Conv layers)

INPUT/OUTPUT DIMENSIONS

2. Pooling Layer

Let, Input Dimensions → **[W_{in}, H_{in}, C_{in}]**

Output Dimensions → **[W_{out}, H_{out}, C_{out}]**

Then,

$$W_{out} = \frac{W_{in} - F + 2*P}{\#Stride} + 1$$

$$H_{out} = \frac{H_{in} - F + 2*P}{\#Stride} + 1$$

$$C_{out} = C_{in}$$

Usually, In Pooling layers,
#Stride = F
In that case,

$$W_{out} = \frac{W_{in} + 2*P}{F}$$

$$H_{out} = \frac{H_{in} + 2*P}{F}$$

$$C_{out} = C_{in}$$

Where, **F** = Kernel Dimension (**F** for an **F*F** Pooling kernel)
P = Padding Length
#Stride = Strides

INPUT/OUTPUT DIMENSIONS

2. Pooling Layer (For Batch Size N)

Let, Input Dimensions → **[N, W_{in}, H_{in}, C_{in}]**

→ Output Dimensions → **[N, W_{out}, H_{out}, C_{out}]**

Then,

$$W_{out} = \frac{W_{in} - F + 2*P}{\#Stride} + 1$$

$$H_{out} = \frac{H_{in} - F + 2*P}{\#Stride} + 1$$

$$C_{out} = C_{in}$$

Usually, In Pooling layers,
#Stride = F
In that case,

$$W_{out} = \frac{W_{in} + 2*P}{F}$$

$$H_{out} = \frac{H_{in} + 2*P}{F}$$

$$C_{out} = C_{in}$$

Where, **F** = Kernel Dimension (**F** for an **F*F** Pooling kernel)
P = Padding Length
#Stride = Strides

INPUT/OUTPUT DIMENSIONS

3. Flattening

Let, Input Dimensions → **[W_{in}, H_{in}, C_{in}]**

Then,

$\boxed{\text{Output Dimensions} \rightarrow \mathbf{W_{in} \times H_{in} \times C_{in} \rightarrow \#output\ units}}$

4. Fully Connected (Dense) Layer

Let, Input Dimensions → **#input units (neurons)**

Then,

$\boxed{\text{Output Dimensions} \rightarrow \mathbf{\#output\ units\ (neurons)}}$

INPUT/OUTPUT DIMENSIONS

3. Flattening (For Batch Size N)

Let, Input Dimensions → **[N, W_{in}, H_{in}, C_{in}]**

Then,

$\boxed{\text{Output Dimensions} \rightarrow [\ N, W_{in} \times H_{in} \times C_{in}\ ] \rightarrow [\ N, \#\text{output units}\ ]}$

4. Fully Connected (Dense) Layer (For Batch Size N)

Let, Input Dimensions → **[N, #input units (neurons)]**

Then,

$\boxed{\text{Output Dimensions} \rightarrow [\ N, \#\text{output units (neurons)}]}$

PARAMETER CALCULATION

1. Convolutional Layer

For a convolutional layer, the number of parameters is determined by the size of the kernels, the number of filters, and the number of input channels.

We can calculate the number of parameters in the following manner:

$$\boxed{\#\text{Parameters} = \{(k_w \times k_h \times C_{in}) + 1\} \times C_{out}}$$

Where:

- k_w = kernel width
- k_h = kernel height
- C_{in} = number of input channels
- C_{out} = number of filters (output channels)

The “+1” accounts for the bias term for each filter.

PARAMETER CALCULATION

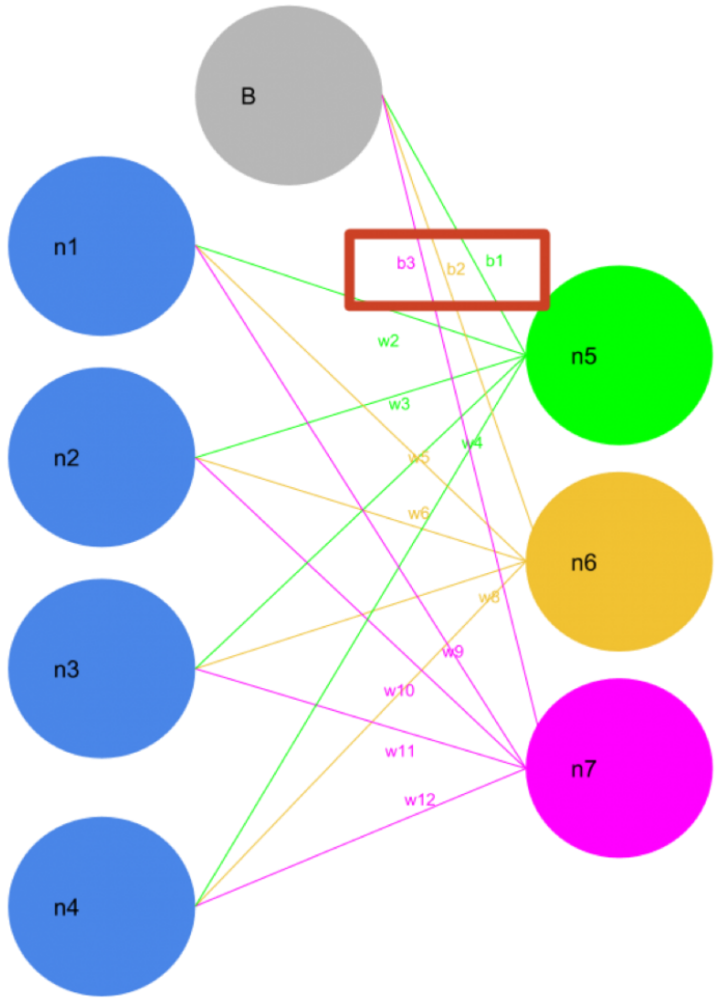
2. Fully Connected (Dense) Layers

For a fully connected layer, the number of parameters is given by the number of input units(neurons) times the number of output units(neurons), plus one bias term for each output unit(neuron).

#Parameters = (#input units + 1) × #output units

Let's take an example, where the fully connected layer has **128 input layer units** and **64 output layer units.** The computed parameters are:

#Parameters = (128 +1) × 64 = 129 × 64 = 8256



PARAMETER CALCULATION

3. Pooling Layers

Pooling layers (e.g., max pooling, average pooling) do not have learnable parameters, so they contribute **0** to the parameter count.

4. Batch Normalization Layers

For batch normalization layers, each feature channel has two parameters (**gamma** and **beta**).

$\boxed{\textbf{\#Parameters} = \mathbf{2 \times C_{out}}}$

If a batch normalization layer is applied after a Fully Connected layer, then,

$\boxed{\textbf{\#Parameters} = \mathbf{2 \times \#output\ units}}$

Let's suppose that the batch normalization layer is applied to an output of **64** channels. Then the number of parameters can be computed as:

#Parameters = 2 × 64 = 128

N.B: In practical scenarios, there are actually **four** parameters (**gamma**, **beta**, and the **moving mean** and **variance**) for BN layers. However, the latter two are **non-trainable parameters.**

EXAMPLE PROBLEMS

EXAMPLE 1

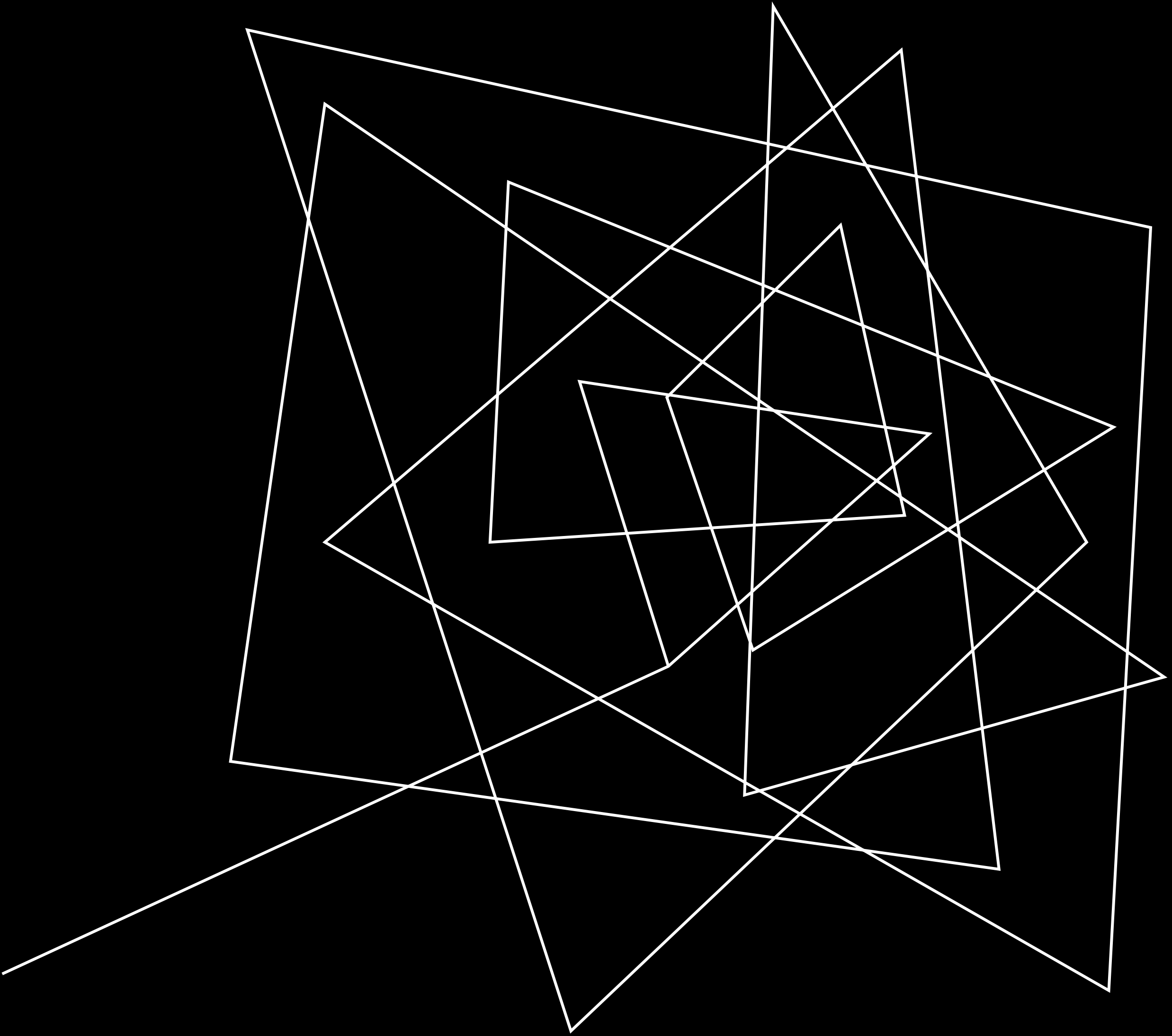
Layer	Input Dimensions	Filter Size	#Filters or, #Neurons	Padding	Output Dimensions	#Params
Conv1	128 * 128 * 3	5*5	32	2		
MaxPool1		2*2	—	0		
Conv2		3*3	64	0		
MaxPool2		2*2	—	0		
Flatten		—	—	—		
FC		—	128	—		
FC (Output)		—	4	—		

The above table represents a CNN architecture and its number of parameters in each layer. **Complete the table** and Comment on the **number of classes.**

EXAMPLE 1

Layer	Input Dimensions	Filter Size	#Filters or, #Neurons	Padding	Output Dimensions	#Params
Conv1	128 * 128 * 3	5*5	32	2		
MaxPool1		2*2	—	0		
Conv2		3*3	64	0		
MaxPool2		2*2	—	0		
Flatten		—	—	—		
FC		—	128	—		
FC (Output)		—	4	—		

Refer to the same table and **determine the dimension of the receptive field of a neuron in the Convolutional layer 2 (Conv2)**. What would be the dimensions **if the Max Pooling layer 1 (MaxPool1) was removed**?



BACKPROPAGATION IN CNN

BACKPROPAGATION IN CNN

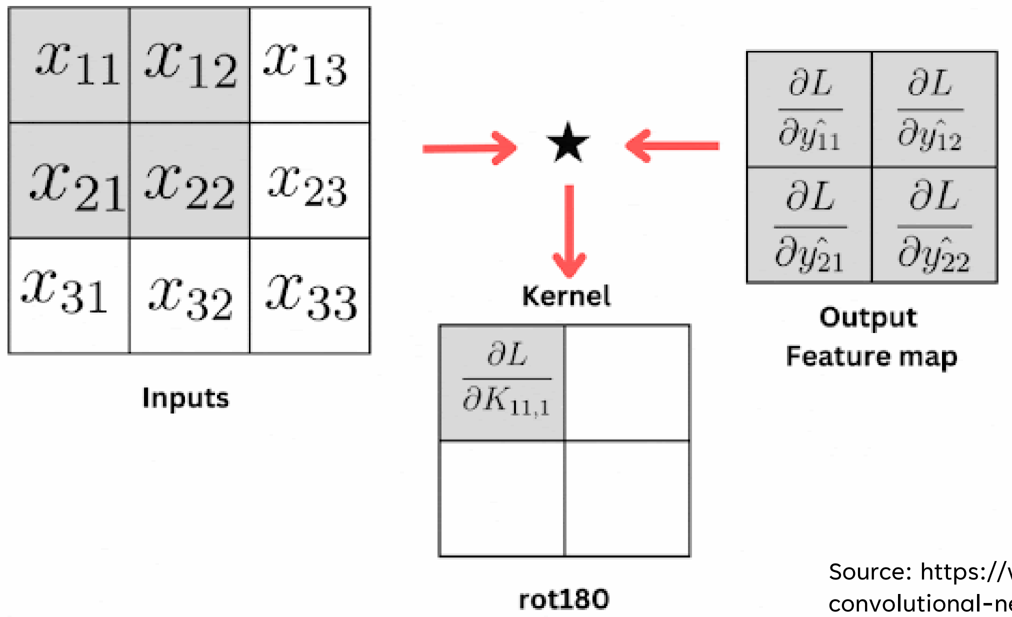
CNNs learn by tweaking the filters so that the final prediction gets better.. just like how MLPs adjust weights. The only difference is **how** the weights are shared and **where** they apply. Naturally, updating weights of the convolutional layers (filter values) also must consider **where that specific filter applies**.

Just like MLPs (and dense layers), **Backpropagation** is used to update filter weights in CNNs too. The loss is used to **compute gradients** of the weights in all layers, and just like MLPs, **chain rule is applied layer-by-layer, in reverse order, from output to input**. However, there are some differences too:

Aspect	CNNs	MLPs
Weight Sharing	Same filter used across different locations → **fewer parameters.**	Each weight is unique.
Local Connectivity (Sparse vs. Dense)	Neurons connect to a small region (**Sparse Connectivity**).	Neurons connect to all inputs.
Gradient Computation	Gradients are **aggregated** for **each filter** across **all locations**.	Each weight has its own unique gradient.

BACKPROPAGATION IN CNN

As we can see, in CNNs, each filter is reused across the image. So, we calculate **how each instance of that filter affects the loss** and update it accordingly. Backprop must tell us how to adjust the filter based on all the places we used it on — not just one. This is actually done **using another convolution-like operation with the flipped filter, which aggregates gradients across all locations** — just as the filter was applied repeatedly during the forward pass. We can actually show that the gradients of the kernel are produced by **convolving** the output feature map and the input matrix.



x_{11} x_{12} x_{13}
x_{21} x_{22} x_{23}
x_{31} x_{32} x_{33}
Inputs
$\star$
Kernel
$\frac{\partial L}{\partial K_{11,1}}$
rot180
$\frac{\partial L}{\partial \hat{y_{11}}}$ $\frac{\partial L}{\partial \hat{y_{12}}}$
$\frac{\partial L}{\partial \hat{y_{21}}}$ $\frac{\partial L}{\partial \hat{y_{22}}}$
Output
Feature map

Source: https://www.quarkml.com/2023/07/backward-pass-in-convolutional-neural-network-explained.html

BACKPROPAGATION IN CNN

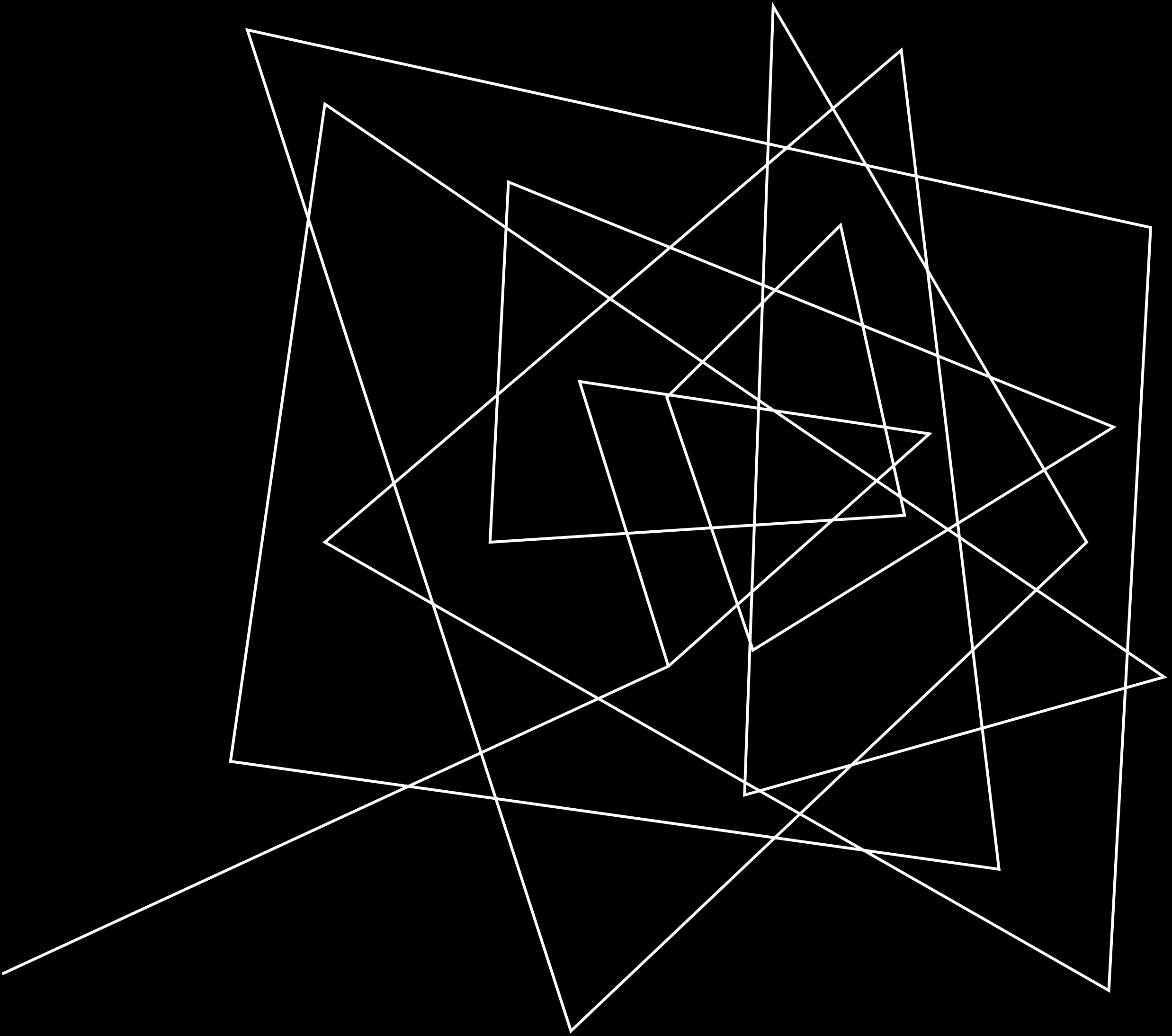
In short, Backpropagation through convolutional layers is another convolution process. The gradients can be found using the following formula:

Finding the gradients:

$$\frac{\partial L}{\partial F} = \text{Convolution}\left(\text{Input } X \text{ , Loss gradient } \frac{\partial L}{\partial O}\right)$$

$$\frac{\partial L}{\partial X} = \begin{matrix}\text{Full}\\ \text{Convolution}\end{matrix}\left(\begin{matrix}180^\circ\text{rotated}\\ \text{Filter } F\end{matrix} , \begin{matrix}\text{Loss}\\ \text{Gradient}\end{matrix}\ \frac{\partial L}{\partial O}\right)$$

We won’t go into the math behind it. But you can go through this article if you are interested:
https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c



CNN ARCHITECTURE DESIGN & MODERN ARCHITECTURES

CNN ARCHITECTURE DESIGN

We have already discussed how a typical CNN architecture looks like. However, a convolutional neural network structure can vary widely (unlimited variations possible). So, it is important to structure a CNN in an intuitive and practical way. Different choices affect model capacity, overfitting, and performance.. So, a CNN must be designed considering all these factors.

General Structure of a CNN

A typical CNN has 3 main parts:

- **Input Layer:** Accepts image of shape: (H × W × C) — e.g., (224×224×3 for RGB).
- **Feature Extraction Block:** Repeating pattern consisting of convolutional blocks: **[Conv → Activation (ReLU) → (Optional BatchNorm) → Pooling].** This block is stacked multiple times, often with increasing filter counts.
- **Classification Head: [Flatten or Global Average Pooling → Dense Layer(s) → Output].** Often ends with SoftMax (classification) or sigmoid (binary).

CNN ARCHITECTURE DESIGN

The table below summarizes the most important design choices, trends and heuristics for convolutional neural networks.

Component	**Key Considerations**
# of Conv Layers	More layers = better abstraction but harder to train.
Filter Size	3×3 is common (captures local patterns); sometimes 1×1 or 5×5.
# of Filters	Increases with depth (e.g., 32 → 64 → 128) to capture complex features.
Padding	'Same' (output same size) vs. 'valid' (shrinks output).
Strides	Controls downsampling speed (use 2 in place of pooling sometimes).
Pooling	Max pooling for sharp features; Avg pooling for smooth ones.
Activation	ReLU is default; alternatives: Leaky ReLU, ELU.
Batch Normalization	Stabilizes training, allows higher learning rate.
Dropout	Reduces overfitting (especially in dense layers).

CNN ARCHITECTURE DESIGN

We have a sample 5-layer CNN architecture for classifying 224x224x3 images into 10 classes on the right. This architecture can be further improved by accommodating batch normalization layer after the convolutional layers.

Designing a CNN is like crafting a visual processing pipeline. Early layers capture details, deeper layers abstract, and the classifier makes the decision. Smart choices in **filter size, layer depth,** and **regularization** make all the difference.

Some Key Design Intuitions

- Early layers learn edges and textures, deeper layers learn object parts.
- Deeper networks are powerful but more prone to vanishing gradients and overfitting.
- Using lobal average pooling instead of flatten+dense can reduce overfitting.

Think: *If you were designing a CNN to classify hand-written digits vs. cancer tumors in MRI scans, how would your architecture differ?*

Input (224x224x3)
↓
Conv (3x3, 32 filters) → ReLU
↓
Max Pooling (2x2)
↓
Conv (3x3, 64 filters) → ReLU
↓
Max Pooling (2x2)
↓
Conv (3x3, 128 filters) → ReLU
↓
Global Average Pooling
↓
Dense (128) → ReLU → Dropout
↓
Dense (10) → SoftMax (Output)

MODERN CNN ARCHITECTURES

Now that we understand how CNNs are built and how they work, let's look at how they've evolved over time through some landmark architectures. **Each of these models brought a new idea** to the table: from **deeper networks** and **smarter connections** to **more efficient use of parameters**. We'll go through a brief timeline — starting from LeNet, the earliest CNN, all the way to efficient and scalable models like MobileNet and EfficientNet. For each, we'll highlight their structure, innovations, and what made them special. These models have been primarily designed for **image classification** tasks, primarily the ImageNet Challenge (A benchmark classification task), but their concepts can be extended beyond classification to other tasks too like image segmentation, object detection, localization, etc.

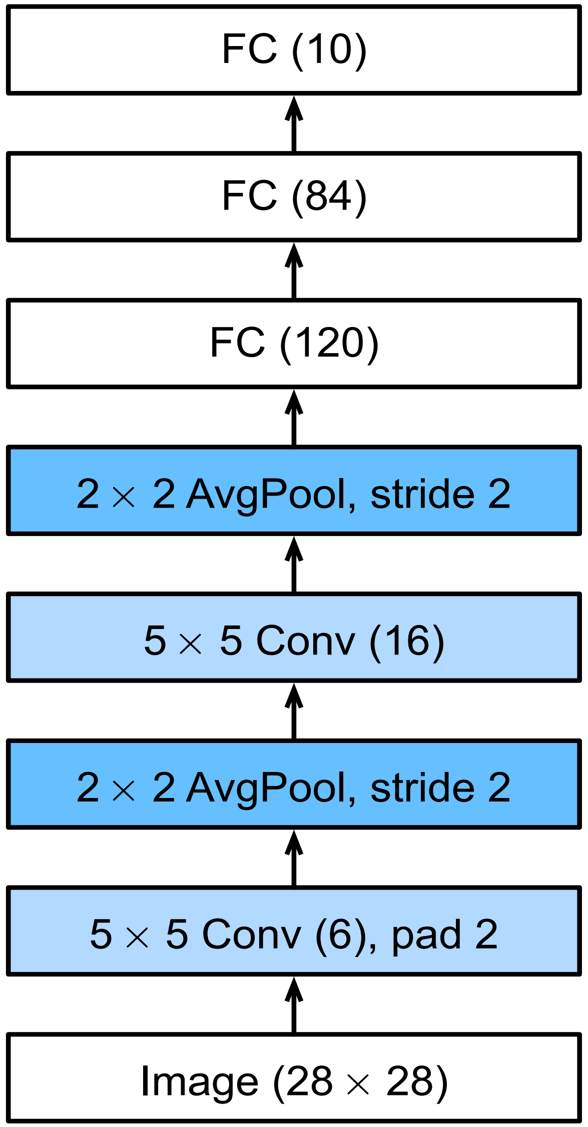
Some of the most popular CNN architectures proposed over the years are:

- LeNet (1998)
- AlexNet (2012)
- VGG (2014)
- GoogLeNet (2014)
- ResNet (2015)
- DenseNet (2017)
- MobileNet (2017)
- EfficientNet (2019)

LENET-5

Before deep learning took off, **LeNet** — a simple yet powerful architecture that laid the groundwork for modern CNNs.
Designed to recognize handwritten digits on checks, LeNet-5 was probably the first network to successfully combine convolution, subsampling (pooling), and fully connected layers.. all trained end-to-end.

The Diagram on the right outlines the architecture of Lenet-5. The architecture consists of two Conv. + AvgPooling blocks followed by flattening and 3 dense layers (including Output). It was proposed by Yann LeCun et al. back in 1998, kickstarting the era of CNNs.



By Zhang, Aston and Lipton, Zachary C. and Li, Mu and Smola, Alexander J. - https://github.com/d2l-ai/d2l-en, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=152265656

LENET-5

Key Features:

- **First successful CNN** used in a real-world application: digit recognition by banks.
- Introduced **parameter sharing and local receptive fields**.
- Used **tanh activation** (ReLU wasn’t popular yet).
- Used **average pooling** instead of max pooling.
- Only partially connected in some layers (C3) to reduce computation.

Takeaways:

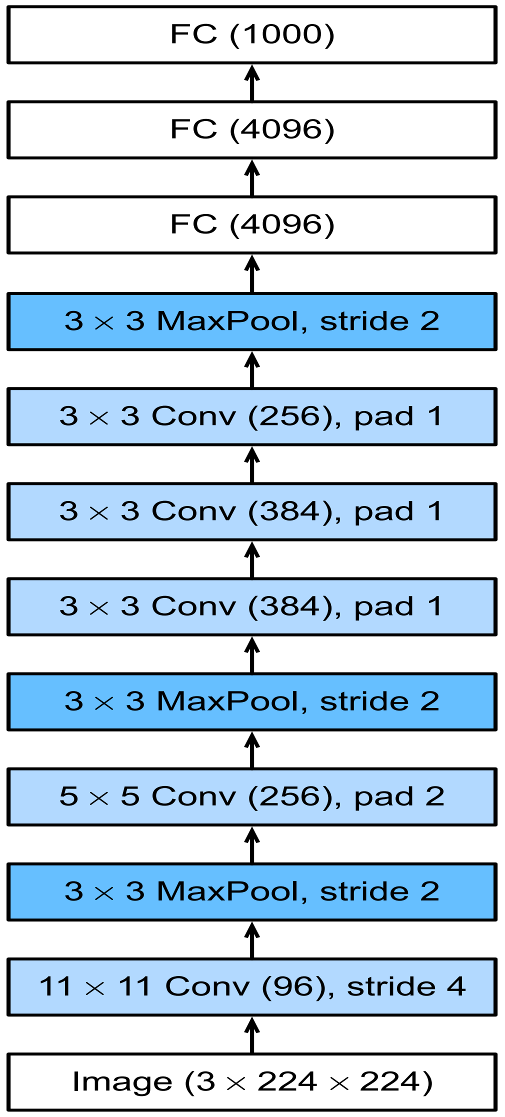
- LeNet-5 was the foundation for all future CNNs.
- Demonstrated that CNNs can capture spatial hierarchies better than MLPs for image tasks.
- Set the stage for deeper and more complex architectures (like AlexNet).

LeNet-5 might seem small by today’s standards, but it taught us **a powerful idea**: **CNNs can learn spatial hierarchies of features, from edges to digits** — automatically.

ALEXNET

LeNet proved that CNNs work, but it was **AlexNet** that proved they **scale**. In 2012, AlexNet dramatically outperformed all other models on the ImageNet classification challenge, marking the beginning of the deep learning revolution in computer vision. This model was deeper, wider, and smarter.. and **made GPUs mainstream in DL training**.

The Diagram on the right outlines the architecture of AlexNet. The architecture consists of 5 Conv. Layers (11×11, 5×5, and three layers with 3×3 filters) followed by flattening and 3 dense layers (including Output). It used Max Pooling instead of Avg Pooling. It was developed in 2012 by Alex Krizhevsky in collaboration with Ilya Sutskever and his Ph.D. advisor Geoffrey Hinton (Know him?). It was trained using two GPUs in parallel.



By Zhang, Aston and Lipton, Zachary C. and Li, Mu and Smola, Alexander J. - https://github.com/d2l-ai/d2l-en, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=152265656

ALEXNET

Innovations:

- **ReLU activation introduced** — much faster training than tanh/sigmoid.
- **Dropout introduced** — to reduce overfitting in fully connected layers.
- Trained using two GPUs in parallel.
- **Introduced Data Augmentation** (translations, reflections) + Local Response Normalization (LRN).
- Made deep architectures practical — 60 million parameters trained on ImageNet!

AlexNet didn’t invent CNNs — but it made them **famous**. It showed the world that **deep learning + GPUs = unbeatable visual recognition**. AlexNet alone is perhaps responsible for most of the common practices we follow in deep learning nowadays. Even the future “Better” architectures was actually built on the success of AlexNet.

VGGNETS (VGG-16 & VGG-19)

After AlexNet and ZFNet, researchers asked: **Can we improve accuracy by going deeper?** Without complicating the architecture? **VGG** answered with a bold 'yes' by stacking simple 3×3 convolution layers — no tricks, just **depth**. It showed that depth and simplicity could outperform clever tweaks, and it remains a baseline architecture even today.

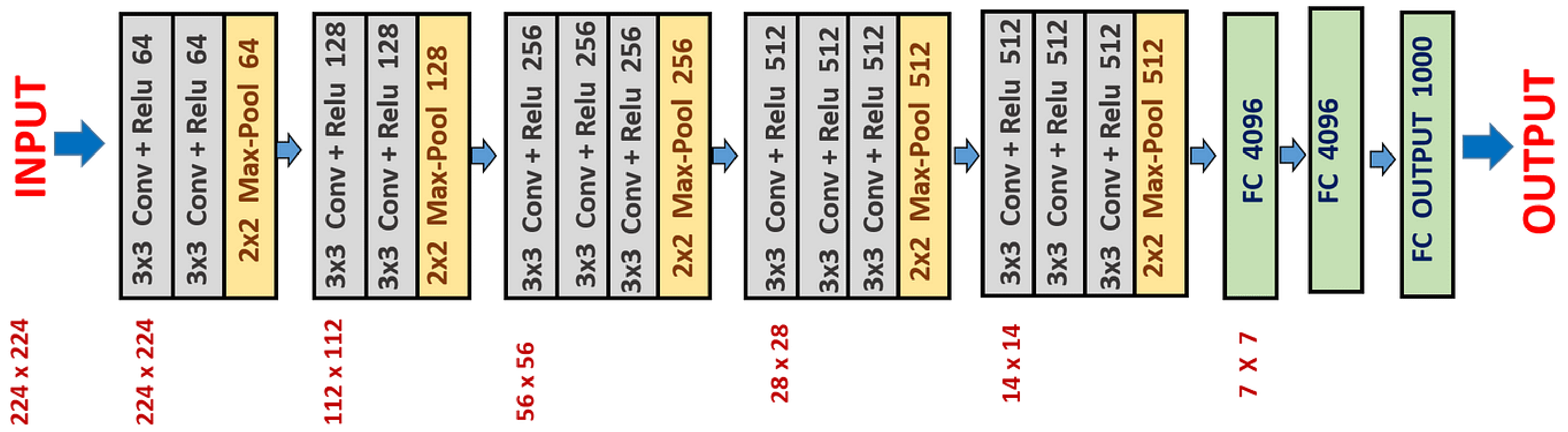
Key Architecture Concepts:

- Uses only **3×3 convolutions**, stride 1, padding 1.
- Stacks multiple conv layers before max-pooling.
- Consistent use of **2×2 max pooling with stride 2**.
- **Increased depth** — up to 19 layers (in VGG-19).
- Uses ReLU activations and fully connected layers at the end.
- Trained on ImageNet, achieving top-5 error of **7.3%.**

VGG taught us that **depth matters** — and that **small filters, when stacked, can model large receptive fields.**

VGG-16 & VGG-19

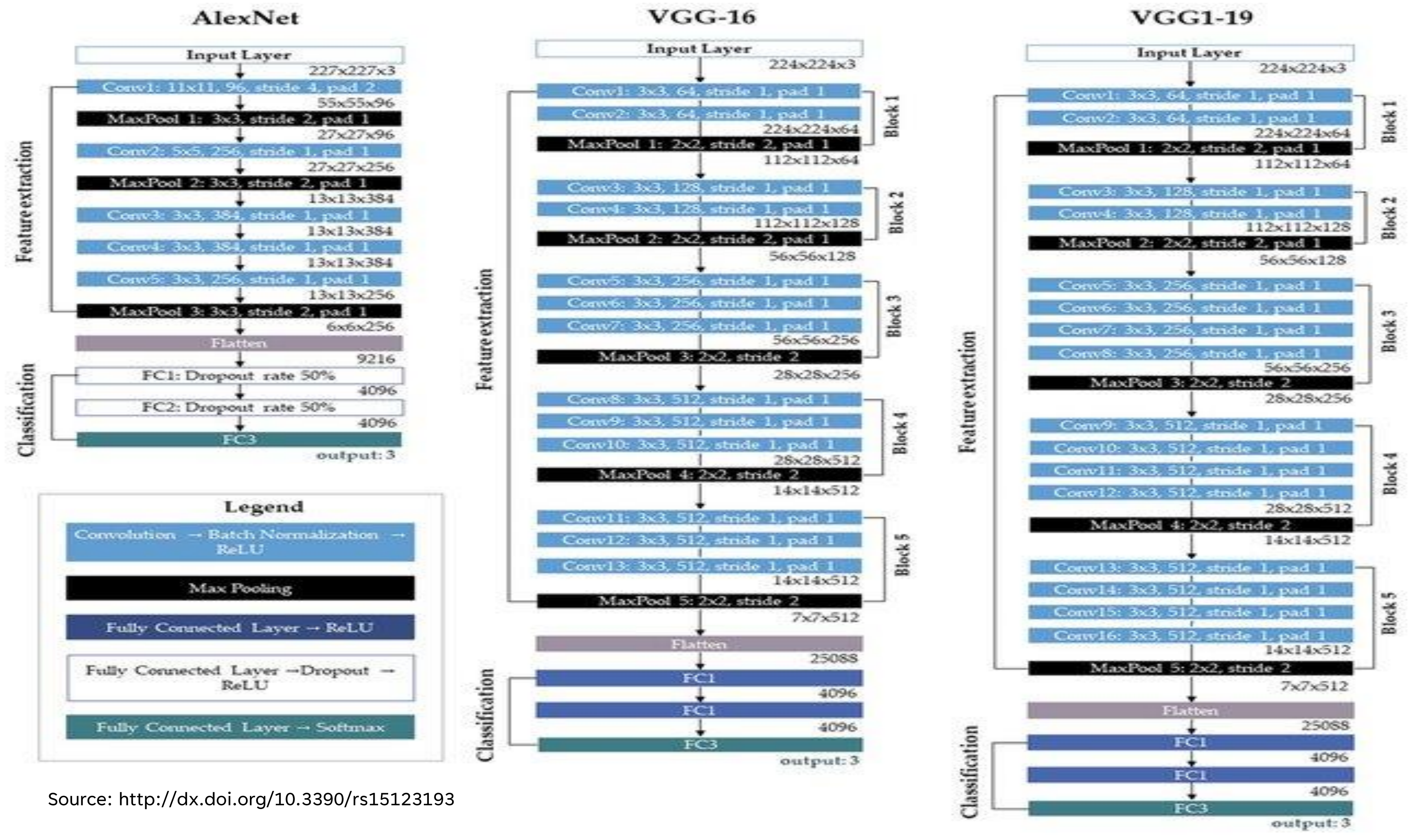
The VGG-16 architecture consists of 13 Conv. Layers (all 3×3 filters) followed by flattening and 3 dense layers (including Output). It was (along with VGG-11 and VGG-19) developed in 2014 by the Visual Geometry Group (VGG) at the University of Oxford. The following diagrams outlines its structure. VGG-19 adds 1 conv. layer each to the latter 3 convolutional blocks.



Source: https://www.kaggle.com/code/thepinokyo/fruit-detection-with-vgg-16

VGG-16 & VGG-19

Observe how VGG-16 and VGG-19 increased the depth of CNNs compared to AlexNet.



AlexNet
Input Layer
227x227x3
Conv1: 11x11, 96, stride 4, pad 2
55x55x96
MaxPool 1: 3x3, stride 2, pad 1
27x27x96
Conv2: 5x5, 256, stride 1, pad 1
27x27x256
MaxPool 2: 3x3, stride 2, pad 1
13x13x384
Conv3: 3x3, 384, stride 1, pad 1
13x13x384
Conv4: 3x3, 384, stride 1, pad 1
13x13x384
Conv5: 3x3, 256, stride 1, pad 1
13x13x256
MaxPool 3: 3x3, stride 2, pad 1
6x6x256
Flatten
9216
FC1: Dropout rate 50%
4096
FC2: Dropout rate 50%
4096
FC3
output: 3
Feature extraction
Classification
Legend
Convolution → Batch Normalization → ReLU
Max Pooling
Fully Connected Layer → ReLU
Fully Connected Layer → Dropout → ReLU
Fully Connected Layer → Softmax
VGG-16
Input Layer
224x224x3
Conv1: 3x3, 64, stride 1, pad 1
Conv2: 3x3, 64, stride 1, pad 1
224x224x64
MaxPool 1: 2x2, stride 2, pad 1
112x112x64
Block 1
Conv3: 3x3, 128, stride 1, pad 1
Conv4: 3x3, 128, stride 1, pad 1
112x112x128
MaxPool 2: 2x2, stride 2, pad 1
56x56x128
Block 2
Conv5: 3x3, 256, stride 1, pad 1
Conv6: 3x3, 256, stride 1, pad 1
Conv7: 3x3, 256, stride 1, pad 1
56x56x256
MaxPool 3: 2x2, stride 2
28x28x256
Block 3
Conv8: 3x3, 512, stride 1, pad 1
Conv9: 3x3, 512, stride 1, pad 1
Conv10: 3x3, 512, stride 1, pad 1
28x28x512
MaxPool 4: 2x2, stride 2
14x14x512
Block 4
Conv11: 3x3, 512, stride 1, pad 1
Conv12: 3x3, 512, stride 1, pad 1
Conv13: 3x3, 512, stride 1, pad 1
14x14x512
MaxPool 5: 2x2, stride 2
7x7x512
Block 5
Flatten
25088
FC1
4096
FC1
4096
FC3
output: 3
Feature extraction
Classification
VGG1-19
Input Layer
224x224x3
Conv1: 3x3, 64, stride 1, pad 1
Conv2: 3x3, 64, stride 1, pad 1
224x224x64
MaxPool 1: 2x2, stride 2, pad 1
112x112x64
Block 1
Conv3: 3x3, 128, stride 1, pad 1
Conv4: 3x3, 128, stride 1, pad 1
112x112x128
MaxPool 2: 2x2, stride 2, pad 1
56x56x128
Block 2
Conv5: 3x3, 256, stride 1, pad 1
Conv6: 3x3, 256, stride 1, pad 1
Conv7: 3x3, 256, stride 1, pad 1
Conv8: 3x3, 256, stride 1, pad 1
56x56x256
MaxPool 3: 2x2, stride 2
28x28x256
Block 3
Conv9: 3x3, 512, stride 1, pad 1
Conv10: 3x3, 512, stride 1, pad 1
Conv11: 3x3, 512, stride 1, pad 1
Conv12: 3x3, 512, stride 1, pad 1
28x28x512
MaxPool 4: 2x2, stride 2
14x14x512
Block 4
Conv13: 3x3, 512, stride 1, pad 1
Conv14: 3x3, 512, stride 1, pad 1
Conv15: 3x3, 512, stride 1, pad 1
Conv16: 3x3, 512, stride 1, pad 1
14x14x512
MaxPool 5: 2x2, stride 2
7x7x512
Block 5
Flatten
25088
FC1
4096
FC1
4096
FC3
output: 3
Feature extraction
Classification

Source: http://dx.doi.org/10.3390/rs15123193

LIMITATIONS OF VGG-16 & VGG-19

Despite the revolutionary performance gains, VGG-16 and VGG-19 had some glaring limitations. Such as:

- VGG-16 has 138 million parameters. VGG-19 has even more, 144 million parameters. Clearly, these models were quite **heavy**. They were also **memory-hungry**. So, computational cost and training was a challenge.
- VGGs featured no skip connections or normalization layers. The models were very deep but really **hard to train without regularization**. The deep architecture made the models somewhat prone to overfitting in the absence of proper regularization.

Despite the limitations, VGG-16 and VGG-19 showed the way to utilize depth to capture large receptive fields and model complex visual recognition. They were the pioneers of **deep** neural networks until ResNet came up with the breakthrough to take the models from deep to **very deep.**

GOOGLENET (INCEPTION)

After VGG's brute-force depth, researchers asked: Can we go deeper without exploding the number of parameters? **GoogLeNet** answered this by introducing the **Inception module**, letting the network **learn what size filters to use** — all in parallel.

Key Concepts:

- Introduced the **Inception** module, combining multiple filter sizes (1×1, 3×3, 5×5) and pooling in parallel.
- Used 1×1 convolutions as bottlenecks to **reduce dimensionality** before costly operations — a major efficiency trick.
- Introduced **global average pooling** instead of fully connected layers at the end → massive reduction in parameters.
- **Deeper** than VGG (**22 layers**), but only **~5 million** parameters.

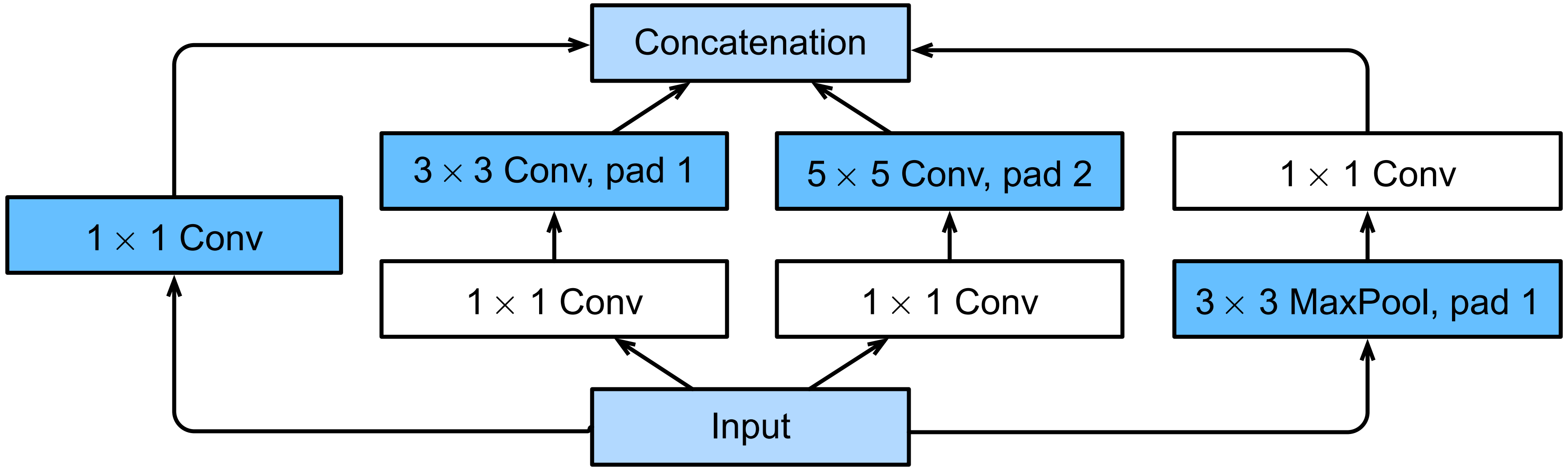
GoogLeNet showed that you don't need to go big to go deep. It introduced a modular way to build deep networks that were both wide and efficient. The Inception idea inspired a whole family of models and industry adoption.

THE INCEPTION MODULE

The core idea behind GoogLeNet was the **Inception** module. Each module includes:

- **1×1 conv →** captures local interactions + reduces dimensions (by **reducing channels**).
- **3×3 conv →** (often preceded by 1×1).
- **5×5 conv →** (preceded by 1×1 to reduce cost).
- **3×3 Max-pooling →** followed by 1×1 conv (to reduce depth).

All these outputs are **concatenated along the depth** dimension. This **enables the model to "decide" the best filter scale for a given feature map region**.



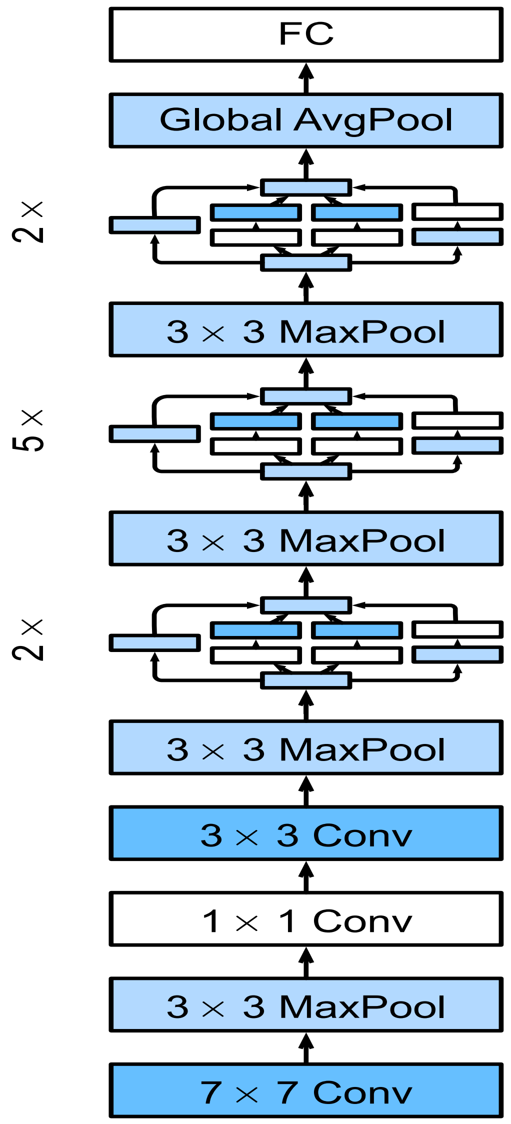
Source: https://d2l.ai/chapter_convolutional-modern/googlenet.html

GOOGLENET (INCEPTION)

The diagram on the right depicts the GoogLeNet (Inception V1) architecture. Several upgraded version have been proposed based on the inception module, getting deeper and deeper and with better performance.

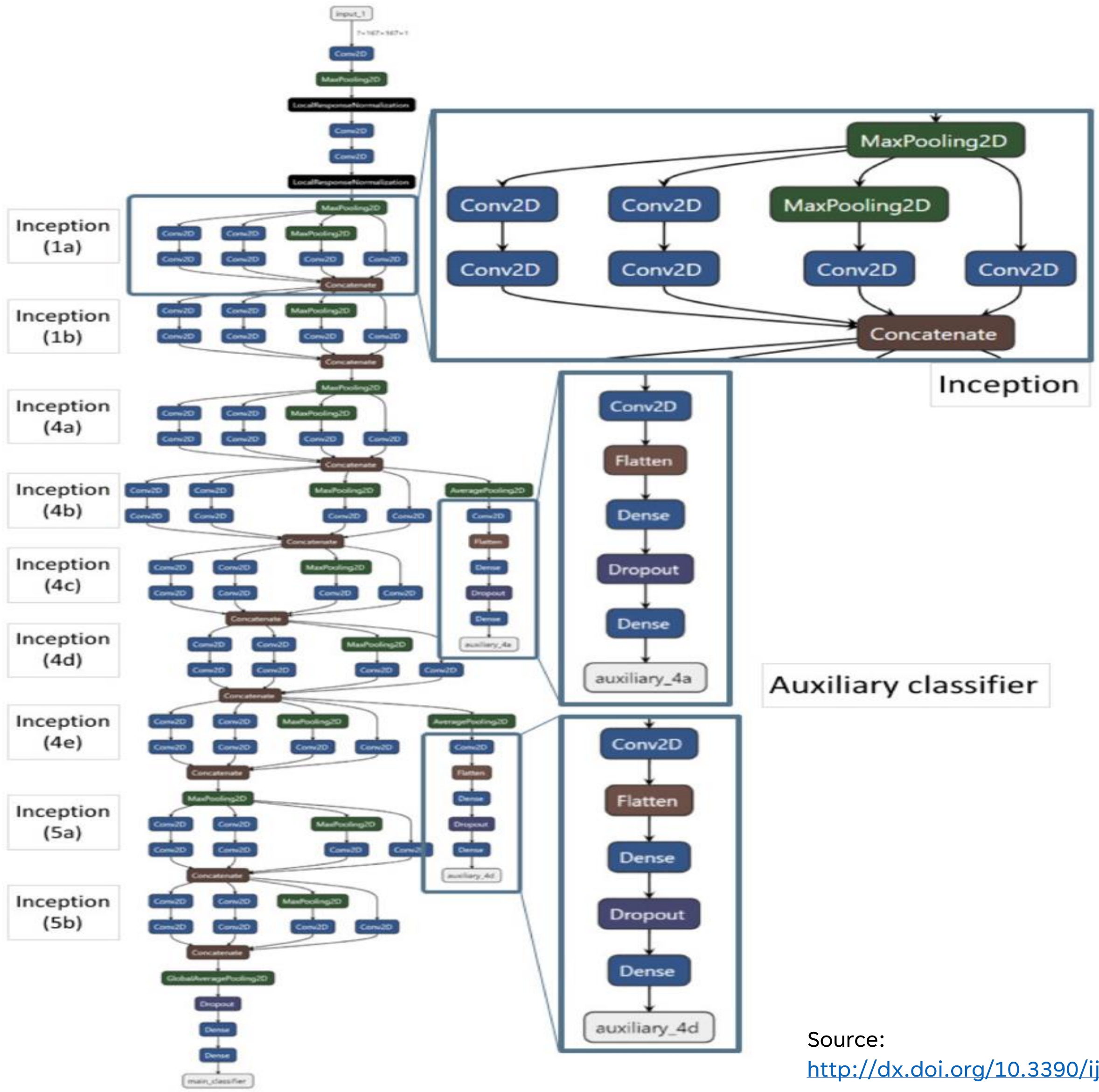
Parameter Efficiency: One of the key features of GoogLeNet. I has ~5 million parameters only (compared to VGG-16's 138M!). Thanks to **1×1 convolutions** and **removal of FC layers**.

Auxiliary Classifiers (Regularization): Two intermediate classifiers were added during training (after Inception 4a and 4d) to help gradient flow and combat vanishing gradients. Each has: Avg Pool → 1×1 Conv → FC → SoftMax. **They basically generate intermediate outputs before reaching the end of the network and help the training process.** *Used only during training (like deep supervision).*



GOOGLENET (INCEPTION)

The **parallel flow of the information** is another key factor in the success of GoogLeNet. Patterns with different scales get captured and stacked on top of each other. A **parallel backpropagation** is also efficient compared to a sequential order since the gradients do not suffer vanishing/exploding. 1x1 convolutions control the feature map sizes. The diagram on the right is another full-fledged variation of GoogLeNet.



Source: http://dx.doi.org/10.3390/ijms22147721

RESNET

As networks got deeper, accuracy plateaued.. and then started getting worse. **Adding more and more layers was no longer improving the performance**. Deep neural networks weren't getting any deeper. ResNet solved this problem by asking: What if instead of learning everything from scratch, the network just learned the '**residual**' — **the part that's different from the input**? ResNet coined in the concept of **Skip connections**, and it was revolutionary.

Key Concepts:

- Instead of learning a direct mapping:

$$H(x) = \textbf{desired output}$$

- ResNet reformulates it as:

$$F(x) = H(x) - x \quad \Rightarrow \quad H(x) = F(x) + x$$

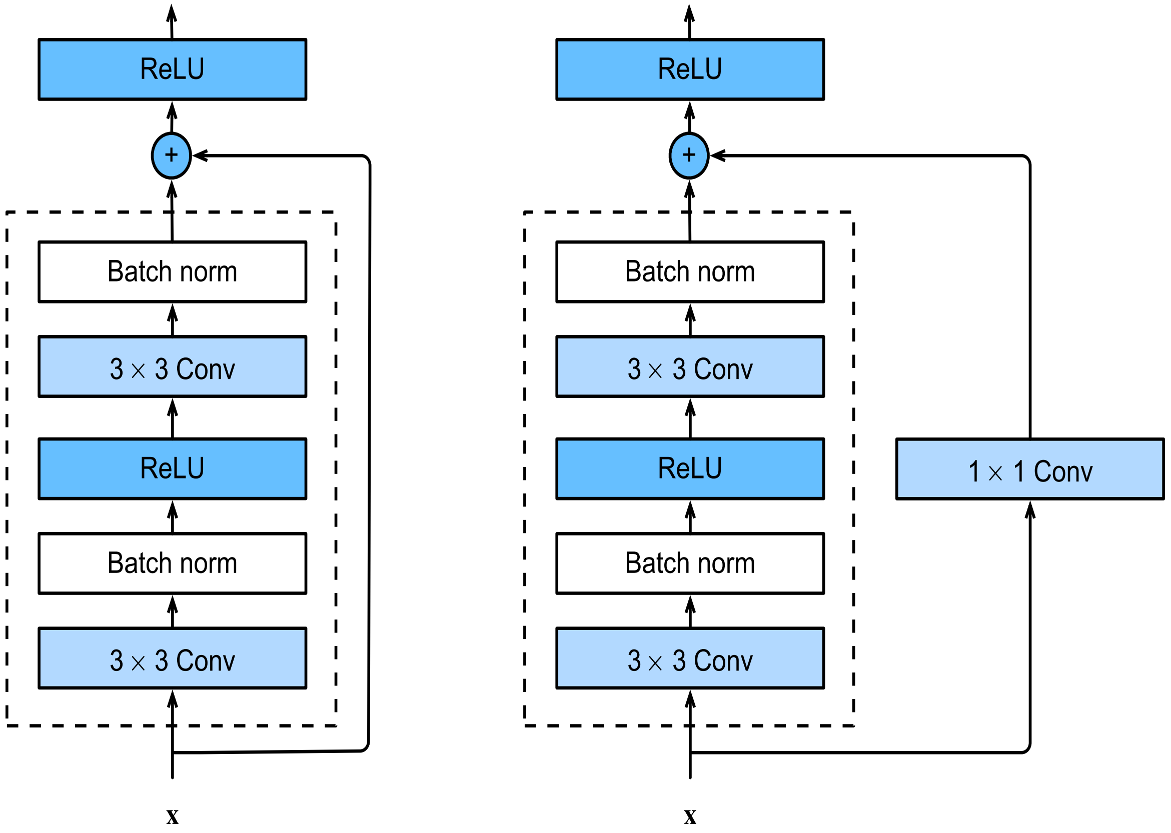
- This means: the network learns the **residual function $F(x)$** and **adds back the input x** through a **skip connection** (also called a shortcut connection).

SKIP CONNECTIONS

Skip Connections are the backbone of ResNet. They solved the Vanishing Gradient problem which was preventing the deeper neural networks from performing well.

- **Identity shortcut:** Passes the input **directly across layers.**
- **Addition operation:**

  Output = F(x,{Wi}) + x
- **Ensures gradients can flow backward** easily, fixing the vanishing gradient problem in deep networks.

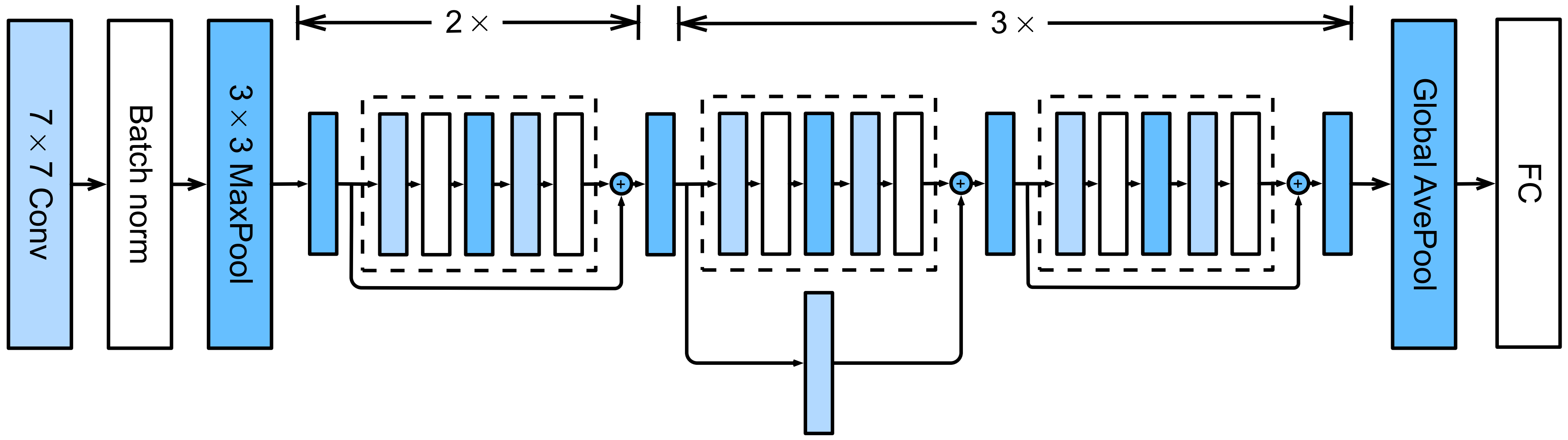


Source: https://d2l.ai/chapter_convolutional-modern/resnet.html

ResNet Block variations with Skip Connections

RESNET

Several ResNet architectures have been proposed over the years based on the base ResNet block. **ResNet-18** was the simplest one among them. There are four convolutional layers in each module (excluding the **1×1** convolutional layer). Together with the first **7×7** convolutional layer and the final fully connected layer, there are 18 layers in total (hence ResNet-18). By configuring different numbers of channels and residual blocks in the module, several different ResNet models were proposed, such as the deeper 152-layer **ResNet-152**.



ResNet-18 Architecture

RESNET

The Table below summarizes different versions of ResNet.

Model	Layers	Notes
ResNet-18	18	2-layer residual blocks.
ResNet-34	34	Deeper version with same block.
ResNet-50	50	First variation to use bottleneck (3-layer blocks).
ResNet-101	101	Deeper version of ResNet-50.
ResNet-152	152	Extremely deep; still trainable!

Why ResNet Works:

Challenge	ResNet's Solution
Vanishing gradients in deep networks.	**Skip connections** to preserve gradients.
Harder to train deeper models.	**Residuals** are easier to optimize.
Overfitting with large models.	Can go deep with **fewer parameters.**
Scalability for different tasks.	Extremely scalable for many tasks: Classification, Detection, Segmentation.

DENSENET

After ResNet revolutionized deep learning with skip connections, researchers asked: what if we pushed this idea even further? Enter **DenseNet** — a network where **each layer connects to every other layer directly** in a feed-forward fashion. This architecture ensures **maximum information flow**, encourages **feature reuse**, and enables **efficient training** even for very deep models.

The key idea behind DenseNet is **Dense Connectivity**. Each layer **receives inputs from all previous layers** and **passes its feature map to all subsequent layers.** Observe the following structure of CNN layers to understand the idea:

$$\mathbf{x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \ldots}$$

$$\mathbf{x_1 = H_1([x_0])}$$

$$\mathbf{x_2 = H_2([x_0, x_1])}$$

$$\mathbf{x_3 = H_3([x_0, x_1, x_2])}$$

Note that **$[x_0, x_1, x_2, ..]$** represents **concatenation,** not **summation.**

DENSENET

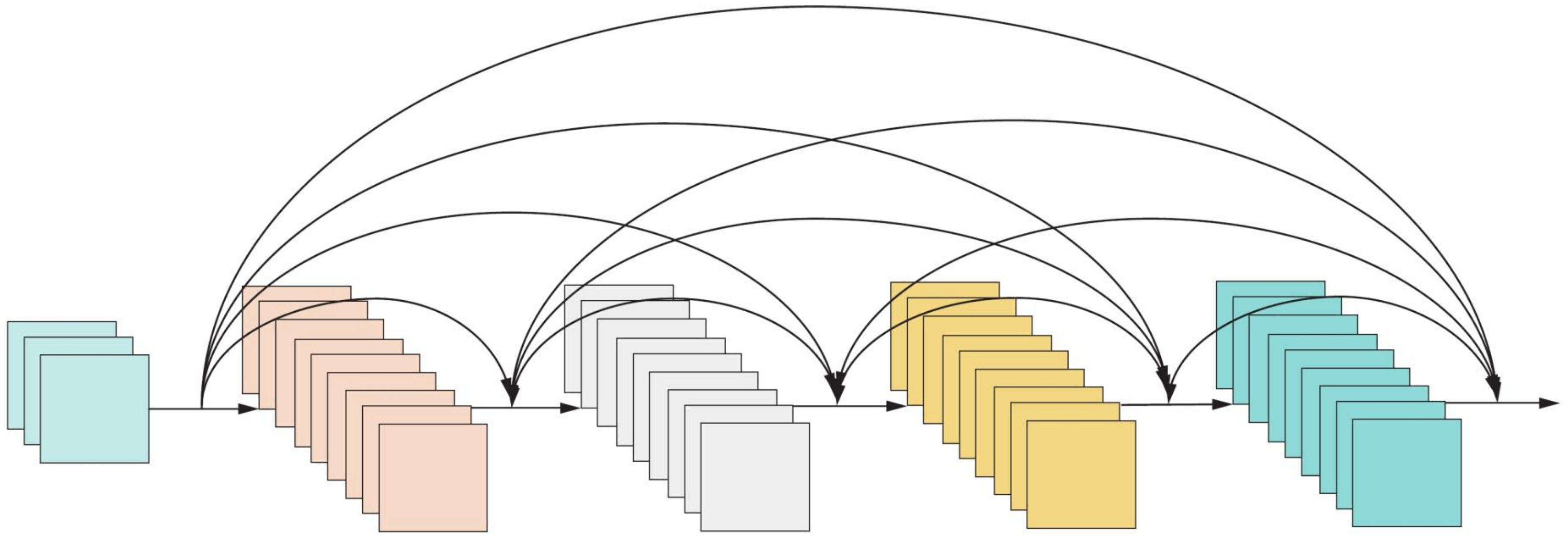
A DenseNet comprises of two foundational components. They are:

1. **Dense Blocks:** A dense block consists of **multiple convolution blocks**, each using the same number of output channels. In the forward propagation, however, we **concatenate the input and output of each convolution block on the channel dimension**. DenseNet uses the modified “batch normalization, activation, and convolution” structure of ResNet.
2. **Transition Layers:** Since **each dense block will increase the number of channels**, adding too many of them will lead to an excessively complex model. **A transition layer is used to control the complexity of the model.** It reduces the number of channels by using a **1×1** convolution. Moreover, it halves the height and width **via average pooling** with a stride of 2.

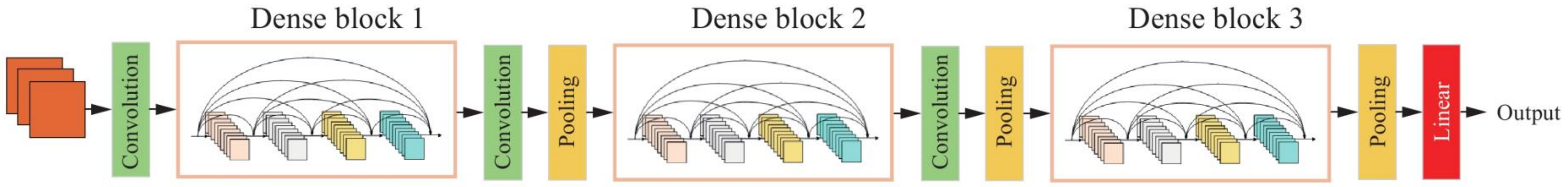
A full-fledged DenseNet first uses the same single convolutional layer and max-pooling layer as in ResNet. Like residual blocks in ResNet, DenseNet uses multiple dense blocks. The number of convolutional layers used in each dense block can be varied. A global pooling layer and a fully connected layer are connected at the end.

DENSENET

The diagrams below depict the DenseNet Architecture. Notice how the Transition layers (**Conv. + Pooling** between the dense blocks) control the complexity of the model.



(a) Dense block structure



(b) DenseNet overall structure

Source: https://dx.doi.org/10.1007/s11633-020-1257-9

RESNET vs DENSENET

The Intuitions behind DenseNet:

Advantage	**Explanation**
Feature Reuse	Later layers reuse earlier features.
Better Gradient Flow	Short paths between loss and all layers.
Parameter Efficiency	Fewer parameters despite being deep (due to transition layers).
No Vanishing Gradients	Every layer gets supervision early on.

DenseNet vs ResNet

- ResNet uses **additive skip connections.**
- DenseNet uses **concatenative feed-forward connections.**
- DenseNet is **more compact**, **faster to train**, but can be **memory intensive.**

DenseNet connects every layer to every other, ensuring maximal feature reuse, excellent gradient flow, and efficient parameter usage. Despite these, **ResNet has been more widely adopted and influential than DenseNet in practical applications** due to its simpler architecture and lower memory overhead, making it easier to scale and deploy in practice.

MOBILENET

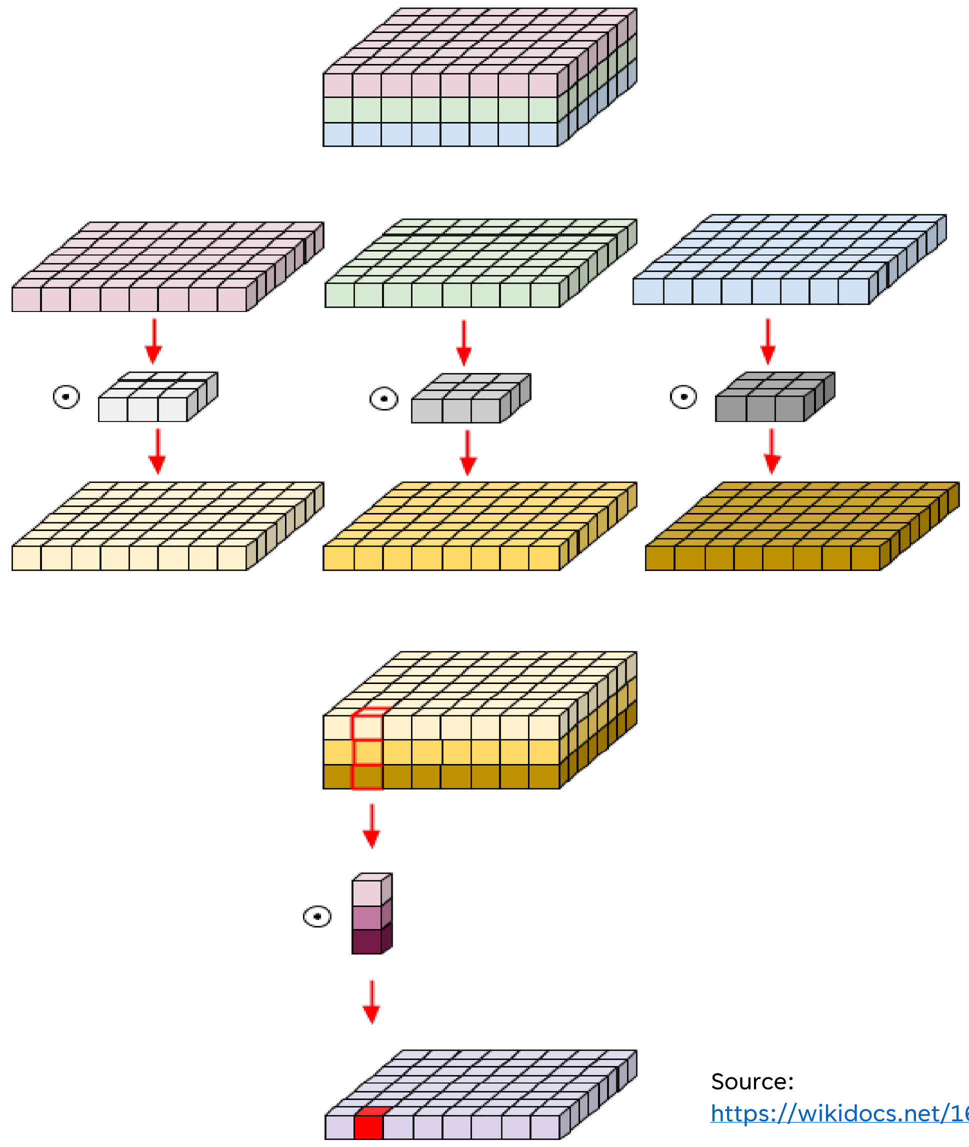
As deep learning models grew deeper and more complex, they became too large and slow for mobile and edge devices. **MobileNet** (by Google, 2017) was designed to address this — providing lightweight, efficient CNNs suitable for real-time applications on phones, drones, and IoT devices.

Key Ideas Behind MobileNet

1. **Depthwise Separable Convolutions:** Instead of using standard convolutions, MobileNet factorizes them into two operations. The first one is **Depthwise convolution**: Applying a single filter per input channel. The second one is **Pointwise convolution**: Using a 1×1 convolution to combine channel outputs. This tweak **reduces computation and parameters by ~8–9× without huge loss in accuracy**.
2. **Width and Resolution Multipliers**: To trade off accuracy for efficiency, MobileNet introduces **Width multiplier (α),** which Shrinks the number of channels. It also introduces **Resolution multiplier (ρ),** which downscales input image resolution. This allows flexibility across devices.

MOBILENET

The diagram on the right shows how **Depthwise Separable Convolution** operates on each channel separately and how 1×1 Pointwise convolutions are used to combine those channel outputs. The applications of MobileNet include Mobile object detection (e.g., TensorFlow Lite apps), Real-time classification on Android/iOS, Embedded vision in smart devices, etc.



Source: https://wikidocs.net/165498

EFFICIENTNET

After years of trial-and-error scaling of CNNs (making them deeper, wider, higher resolution, etc.), researchers asked: What's the optimal way to scale a model efficiently? **EfficientNet** (by Google, 2019) answers this with a principled, compound scaling method.. delivering state-of-the-art accuracy with fewer parameters and FLOPs. EfficientNet is a family of CNNs that are fast, small, and powerful. Instead of blindly making models deeper or wider, it grows the model in a balanced way, kind of like a plant growing both roots and leaves together.

Key Idea Behind EfficientNet

- **Grow Everything Together:** In older models, people made CNNs bigger by:
  - Depth → Add more layers (e.g., going from ResNet-18 to ResNet-101).
  - Width → Use more filters/channels per layer (makes feature maps wider).
  - Resolution → Feed higher-resolution input images (e.g., 224×224 → 380×380).

  But EfficientNet says: Let's grow all three — depth, width, and resolution — at the same time, using a smart rule. Result? Better accuracy, smaller size, faster speed.

EFFICIENTNET

EfficientNet's Solution: Compound Scaling

EfficientNet grows depth, width, and resolution together using **fixed scaling factors**. This is called **Compound Scaling**. We can think of it like resizing a photo: instead of stretching it only vertically or only horizontally, we scale all dimensions proportionally, keeping it sharp and clean. The compound scaling is **based on a small set of constants** (**α** for depth, **β** for width, **γ** for resolution), and a user-specified **scaling coefficient ϕ** to decide how big we want the model.

Why It's Special

Model	**Accuracy (ImageNet)**	**Model Size**	**Speed (FLOPs)**
ResNet-50	~76%	Large	Slower
EfficientNet-B0	~77%	Small	Very Fast
EfficientNet-B7	~85%	Big	Still Efficient

So, even small versions (like B0) **beat older models** in both speed and accuracy. That's why EfficientNet is being used extensively for complex tasks like Medical Imaging.

EFFICIENTNET

The diagram below demonstrating compound scaling has been taken directly from the EfficientNet paper (https://doi.org/10.48550/arXiv.1905.11946).

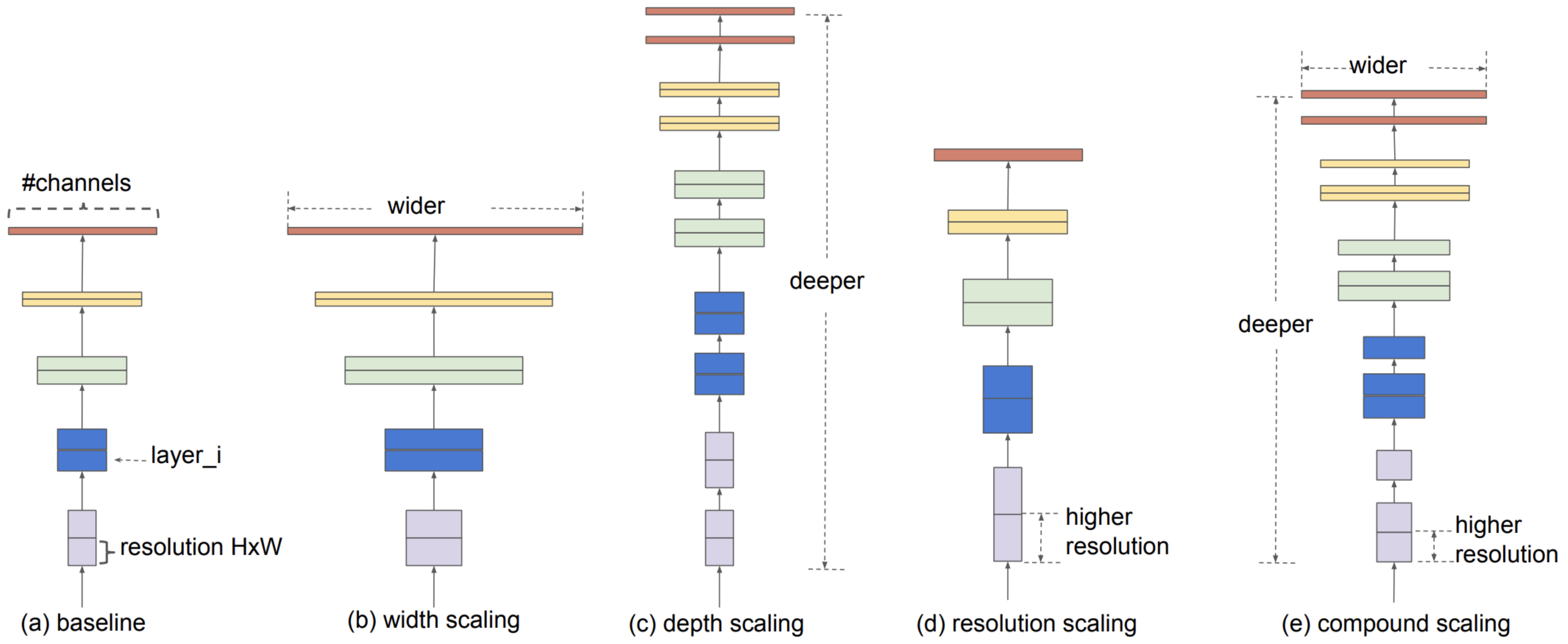
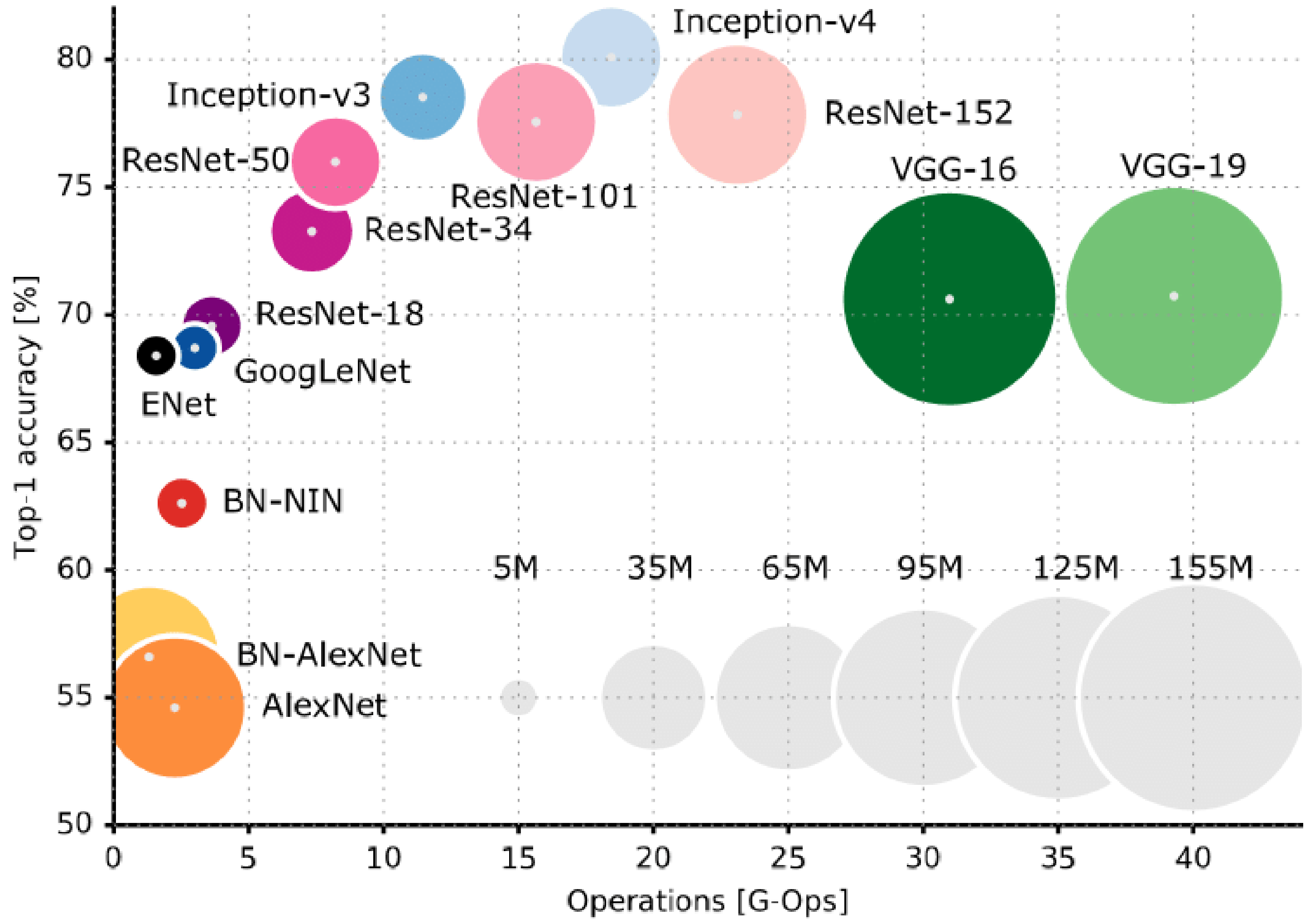


Figure 2. **Model Scaling.** (a) is a baseline network example; (b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution. (e) is our proposed compound scaling method that uniformly scales all three dimensions with a fixed ratio.

COMPARISON BETWEEN CNN MODELS

The figure below depicts the performance of several popular CNN models on ImageNet dataset. It should be noted that **EfficientNet** outperformed all of them achieving an accuracy of **85%.**



COMPARISON BETWEEN CNN MODELS

The key features of all the models discussed have been summarized in the table below.

Architecture	Year	Key Innovation	Depth	Parameters	Strengths
LeNet-5	1998	Early CNN for digit recognition (MNIST)	7	~60K	Simple, foundational, fast
AlexNet	2012	ReLU, dropout, GPU training	8	~60M	Deep learning breakthrough
VGG-16/19	2014	All 3×3 convs, uniform design	16–19	~138M	Simplicity, but very large
GoogLeNet	2014	Inception modules, auxiliary classifiers	22	~6.8M	Efficient depth, less computation
ResNet	2015	Residual connections (skip connections)	18–152	~11M (ResNet-18)	Trains very deep networks successfully
DenseNet	2016	Dense connections between layers	121–201	~8–20M	Strong feature reuse, compact
MobileNet	2017	Depthwise separable convolutions	V1: 28	~4.2M (V1)	Optimized for mobile and edge devices
EfficientNet	2019	Compound scaling (depth+width+res)	B0–B7	5M–66M	State-of-the-art accuracy & efficiency

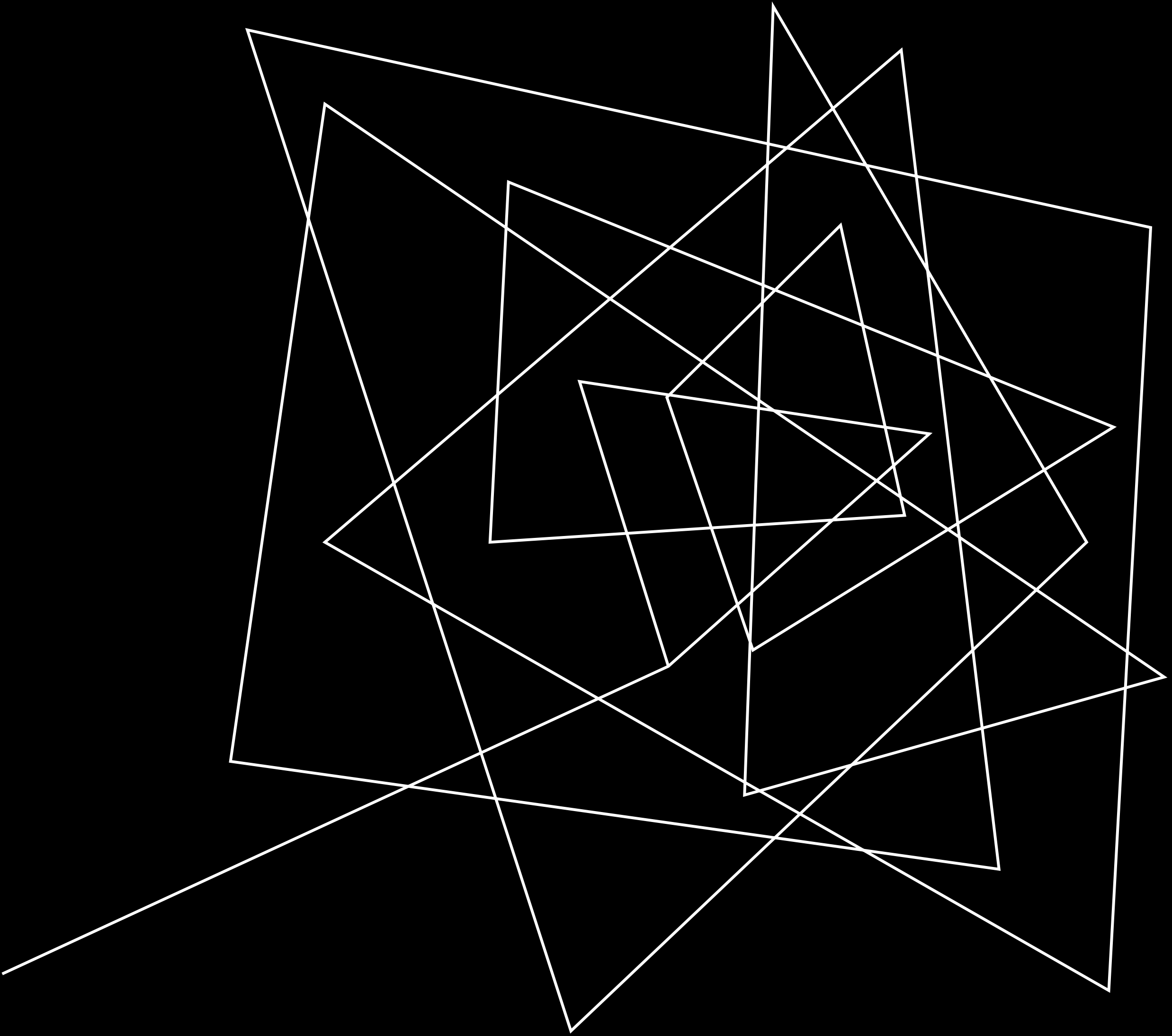


IMAGE CLASSIFICATION & TRANSFER LEARNING

IMAGE CLASSIFICATION WITH CNN

Image classification is one of the most fundamental tasks in computer vision. It involves predicting the class label of an image based on its visual content.

How CNNs Classify Images

- ➢ Learn hierarchical features from raw pixels (Convolutional + Pooling layers).
- ➢ Combine edges → shapes → object parts → full object.
- ➢ Map learned features to class probabilities (Dense layer/layers).

Common Applications

Application	**Description**	**Example Classes**
Object Recognition	Identify the object in the image.	Dog, Cat, Car, Airplane
Medical Imaging	Classify medical scans.	Healthy, Tumor, Pneumonia
Scene Classification	Understand environment context.	Beach, Forest, Street
Fashion Recognition	Identify clothing items.	T-shirt, Sneaker, Dress

Example Dataset Tasks: MNIST (Classify handwritten digits), **CIFAR-10** (10-class object classification), **ImageNet** (1000-class large-scale classification), **Skin Lesion Images** (Benign vs Malignant).

TRANSFER LEARNING

Training a deep CNN from scratch **requires a huge amount of data and computational resources**. But what if we could reuse a model trained on a large dataset for a different (but related) task? That's the essence of transfer learning.

Transfer learning is the process of taking a model that has already learned useful representations from a large dataset (like ImageNet) and applying it to a new problem with less data. For example, **ResNet-50 trained on ImageNet** can be used to classify medical X-ray images. We only have to **retrain the final few layers** on the new dataset. Saves time, needs fewer labeled samples, and still achieves high accuracy.

Why It Works

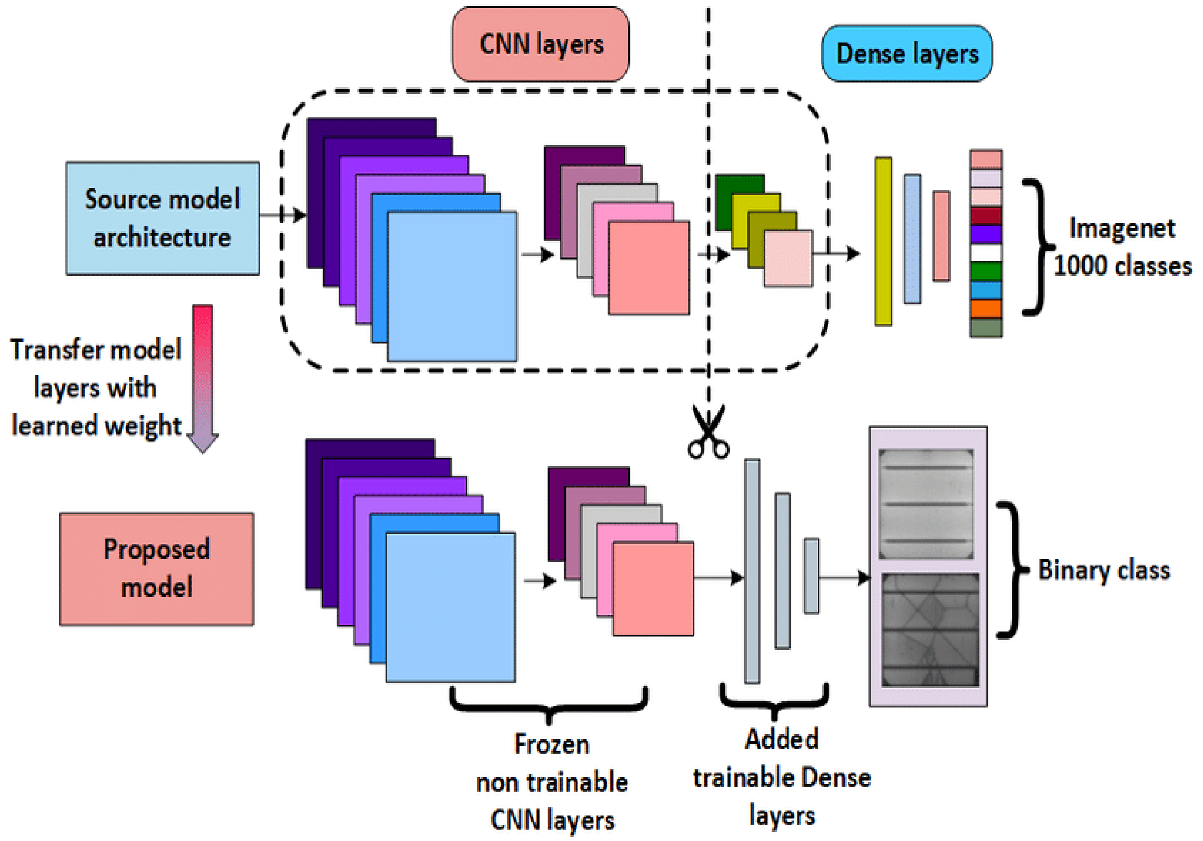
- Early CNN layers capture generic features like **edges**, **blobs**, **textures.** This doesn't vary too much across different tasks and datasets.
- These trained weights are transferable across different image domains.
- Only the final layers (which typically learn task-specific patterns) need modification.

TRANSFER LEARNING

Transfer Learning can be done using two primary strategies. They are:

1. **Feature Extraction:** Use the pre-trained CNN as a fixed feature extractor. Remove **the final classification layer** and train a new one for your task.
2. **Fine-tuning:** Unfreeze **some of the higher (deeper) layers** of the CNN. Continue **training** on your new dataset, allowing the model to adapt the learned features.

VGG, Inception variants, **ResNet, EfficientNet**.. These are some the most suitable and commonly used models for transfer learning. All of them are usually **pretrained on ImageNet**.


CNN layers
Dense layers
Source model architecture
Imagenet 1000 classes
Transfer model layers with learned weight
Proposed model
Binary class
Frozen non trainable CNN layers
Added trainable Dense layers

Source: http://dx.doi.org/10.1109/STI53101.2021.9732592

REFERENCES

1. Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2023). Dive into Deep Learning. Cambridge University Press. Retrieved from https://D2L.ai
2. Stanford cs231n Course Notes (https://cs231n.github.io/)
3. CMU's Introduction to Machine Learning (10-601) Lectures (https://www.cs.cmu.edu/%7Etom/10701_sp11/lectures.shtml)
4. MIT OpenCourseWare: 6.867 Machine Learning (https://people.csail.mit.edu/dsontag/courses/ml16/)
5. University of Toronto - CSC411/2515: Machine Learning and Data Mining (https://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/)
6. Applied ML course at Cornell and Cornell Tech (https://github.com/kuleshov/cornell-cs5785-2020-applied-ml/)
7. https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c
8. https://medium.com/thedeephub/convolutional-neural-networks-a-comprehensive-guide-5cc0b5eae175

THANK YOU