# DEEP LEARNING

# HANDOUT: CNN

## What is CNN (Convolutional Neural Network)?

A **Convolutional Neural Network (CNN)** is a type of **deep learning model** primarily used for **image-related tasks**, but also useful for video, audio, and text data. It mimics how humans perceive visual patterns  focusing on **local features first**, and combining them to understand the whole.

### CNN in Simple Terms

Imagine looking at a picture:

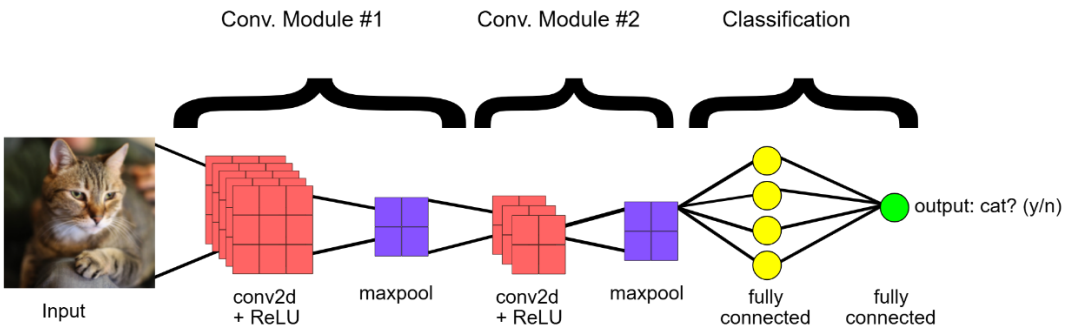
- You first notice **edges and colors**
- Then **shapes**
- Finally, you recognize **objects** like a cat or a car

CNN does the **same thing in layers** — learning:

1. **Edges**
2. **Corners**
3. **Object parts**
4. **Full objects**

## CNN Architecture Components

| Layer | Function |
|---|---|
| 🔹 Convolution Layer | Extracts features (edges, textures) using filters (kernels) |
| 🔹 Activation (ReLU) | Adds non-linearity so model can learn complex patterns |
| 🔹 Pooling Layer | Downsamples image to reduce computation and overfitting |
| 🔹 Fully Connected Layer (FC) | Final decision-making layer (like in regular neural networks) |
| 🔹 Softmax / Sigmoid | Converts outputs to probabilities (for classification) |




### Example: Image of a Dog

| Layer | Output |
|---|---|
| **Input** | 128×128 image of a dog |
| **Conv + ReLU** | Detects edges and fur texture |
| **Pooling** | Shrinks size to retain only important features |
| **More Conv+Pooling** | Detects legs, tail, ears, etc. |
| **Fully Connected Layer** | Combines all to classify as "Dog" or "Not a Dog" |

## 1. Why CNN?

**Scenario: Jungle Surveillance for Camouflaged Objects**

Imagine you're working with a military surveillance team.

**Your task:**
Use drone images to detect hidden enemy tents or equipment camouflaged in dense forest terrain.

### What makes this task hard?

- The objects are **deliberately designed to blend in**.
- The patterns, colours, and textures look almost identical to the natural surroundings.
- Traditional rule-based image processing fails  it **can't generalize** or adapt.

### Why not use a simple image classifier?

Let's say we try using a basic Machine Learning model (e.g., SVM or logistic regression). Here's what goes wrong:

- It **treats the image as flat data** just a giant list of numbers (pixels).
- It **doesn't understand structure**: which pixels are next to each other.

- It's extremely **sensitive to shifts**, rotations, lighting changes.

### Why CNN is a Game-Changer:

| Problem | CNN Solution |
|---|---|
| No spatial understanding | CNN uses **convolution filters** that preserve spatial relations |
| High pixel count | CNN uses **shared weights**, reducing complexity |
| Need for pattern recognition | CNN **learns features** like edges, shapes, textures automatically |
| Camouflaged patterns are subtle | CNN focuses on **local regions** and learns abstract representations |

### Real-World Outcome

Let's say you train a CNN using 1000 drone images labeled "camouflaged" or "natural terrain." After training:

- It learns **visual clues** — like unusual textures, straight edges, or repeating patterns.
- It begins to spot **hidden tarps**, **weapon outlines**, or **tent poles** even if they're mostly covered by leaves.
- It gives you a **probability map** showing regions likely to contain hidden objects.

**"What eyes and brain do together, CNN tries to simulate in software."**
Your eyes look at parts of an image, detect features like edges, textures, and patterns.
Your brain connects those into a meaningful interpretation. CNN mimics that.

## 2. What is an Image and How CNN Reads It

### What is a Digital Image?

A digital image is a **matrix of pixel values**. Each pixel represents the **intensity** (brightness or color) at a particular location.

**Grayscale image** → 2D matrix of values from 0 to 255
(0 = black, 255 = white)

```
[[  0,  34, 128],
 [200, 210, 255],
 [120,  98,  67]]
```

**RGB image** → 3D matrix: Height × Width × 3 channels (Red, Green, Blue)

Example shape: **(224, 224, 3)**

**How CNN Sees an Image**

CNN doesn't "see" like humans. It sees a **tensor** a 3D array of numbers:

**CNN Input Tensor**

```
Height x Width x Channels → (H, W, C)
e.g., (224, 224, 3)
```

Each pixel's color is split into:

- Red intensity (0–255)
- Green intensity (0–255)
- Blue intensity (0–255)


Q. why must the input image be resized to a fixed shape before giving it to a CNN?

**A:** CNN layers require consistent tensor shapes to apply their filters. Without resizing, images of varying size would cause shape mismatches during training.

**Q:** What is the effect of having 3 channels in an image?

**A:** It allows the CNN to separately learn patterns in the Red, Green, and Blue spaces — sometimes objects hide better in one channel than another, especially in camouflage!

**Q:** Why do we normalize pixel values?

**A:** Pixel values in [0–255] can cause unstable training. Normalizing to [0–1] makes learning faster and helps CNN focus on pattern differences, not magnitude.

## 3. Image Preprocessing Steps

Goal: To make every input image clean, standardized, and CNN-ready.

We'll go through the most **commonly used steps**, explain **why we use them**, and show **minimal code with outputs**.

**Scenario: Drone Image of a Jungle Terrain**

Your goal: Preprocess this image so a CNN can later detect **camouflaged objects** (e.g., a hidden tent).

**Step 1: Resize**

**Why?**
CNN architectures (like VGG, ResNet) require a fixed input size (e.g., 224×224). Images from real sources can be anything 480×360, 800×600, etc.

```
from PIL import Image
img = Image.open("jungle_camo.jpg")
img = img.resize((224, 224))
```

**Scenario Insight:**
Without resizing, your CNN won't know how many filters to apply or where it'll crash with shape mismatches.

**Step 2: Convert to RGB (if not already)**

**Why?**
Some images might be grayscale or RGBA. CNNs usually expect **3 channels** (RGB).

```
img = img.convert("RGB")  # Forces RGB
```

**Scenario Insight:**
Camouflaged colors may appear differently in red vs green vs blue channels. This separation helps the CNN.

**Step 3: Convert Image to Numpy Array**

**Why?**
To process pixel values and feed to CNN, we need it as an array.

```
import numpy as np
img_array = np.array(img)  # shape: (224, 224, 3)
```

**Step 4: Normalize Pixel Values (0–255 → 0–1)**

**Why?**
Pixel values range from 0 to 255. High values can slow learning and cause unstable gradients. Normalize to [0, 1].

```
img_array = img_array / 255.0
```

**Scenario Insight:**
Without normalization, the CNN might treat bright leaves (high pixel value) as more "important" than darker camouflaged fabric — which is misleading.

**Step 5 (Optional): Convert to Grayscale**

**Why?**
If you're doing **edge-based or shape-based detection** (not color), grayscale simplifies processing.

```
gray_img = np.dot(img_array[...,:3], [0.2989, 0.5870, 0.1140])
```

**Scenario Insight:**
Grayscale can sometimes enhance visibility of sharp edges — like the outline of a camouflaged weapon.

**Step 6: Histogram Equalization / Contrast Enhancement**

**Why?**
Makes **hidden textures and patterns pop out** — essential in camouflage detection.

**Code (if using scikit-image):**

```
from skimage import exposure
enhanced = exposure.equalize_adapthist(gray_img, clip_limit=0.03)
```

**Scenario Insight:**
Camouflaged cloth might have low contrast. This step helps CNN **see the edges better**.

**Step 7: Convert to CNN Tensor Format**

**Why?**
Keras expects input shape as **(batch, height, width, channels)**. So we expand dims.

```
x_input = np.expand_dims(img_array, axis=0)  # shape: (1, 224, 224, 3)
```

| Step | Purpose | Function/Code |
|---|---|---|
| Resize | Fix input size | img.resize((224,224)) |
| Convert to RGB | Standardize channels | img.convert('RGB') |
| To Array | Process pixels | np.array(img) |
| Normalize | Scale values | / 255.0 |
| Grayscale | Optional, edge tasks | np.dot(..., [0.299, 0.587, 0.114]) |
| Contrast Enhance | Reveal hidden patterns | equalize_adapthist() |
| Expand Dimensions | Tensor format | np.expand_dims(...) |

**Q:** Why is normalization critical for CNNs?

**A:** Without normalization, large pixel values cause inconsistent gradients, poor convergence, and can mislead the model.

**Q:** You have color images of camouflaged tents. Should you convert them to grayscale?

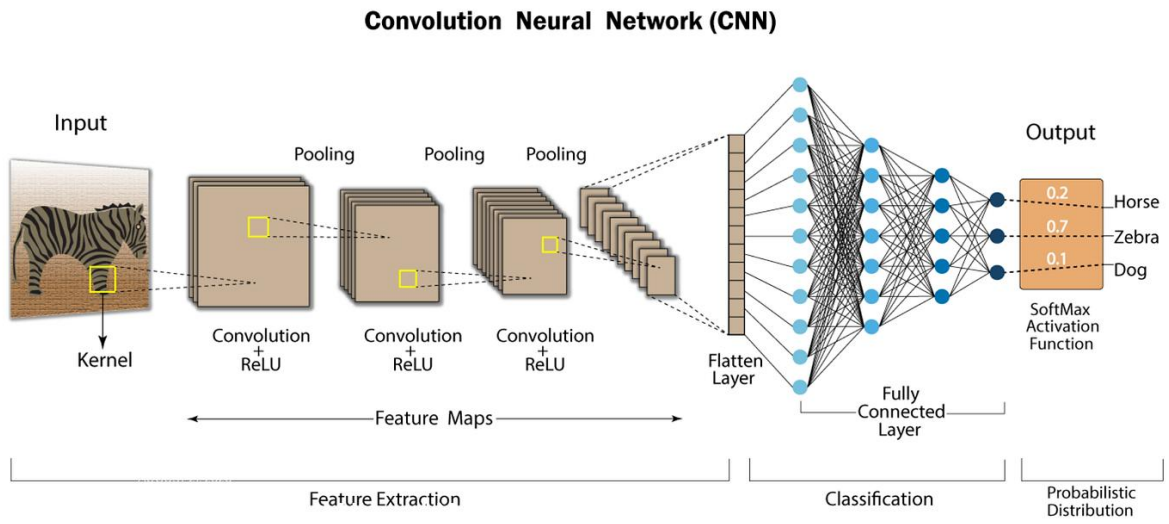
**A:** Only if color doesn't help. If texture/shape is more critical than color, grayscale simplifies the task.

**Q:** What if you skip resizing?

**A:** CNN layers expect fixed-size inputs. Skipping resizing will throw shape errors during training or prediction.

## 4. CNN Architecture

This section explains how a Convolutional Neural Network (CNN) processes an image layer by layer converting pixels into high-level insights such as object presence.




## 1. Input Layer

- The CNN receives a preprocessed image as a tensor.
- Shape: (Height, Width, Channels) → Example: (224, 224, 3) for RGB

**Application Question:**
Why is input shape critical in CNN?
**Answer:** Because filter dimensions and layer configurations depend on consistent input shape; mismatch can break the model.

## 2. Convolution Layer

- This layer applies **filters** (also called kernels) that slide over the image.
- Each filter detects a specific type of feature: edge, texture, color gradient, etc.

**Example:**
A 3×3 filter like

```
[[-1, 0, 1],
 [-2, 0, 2],
 [-1, 0, 1]]
```

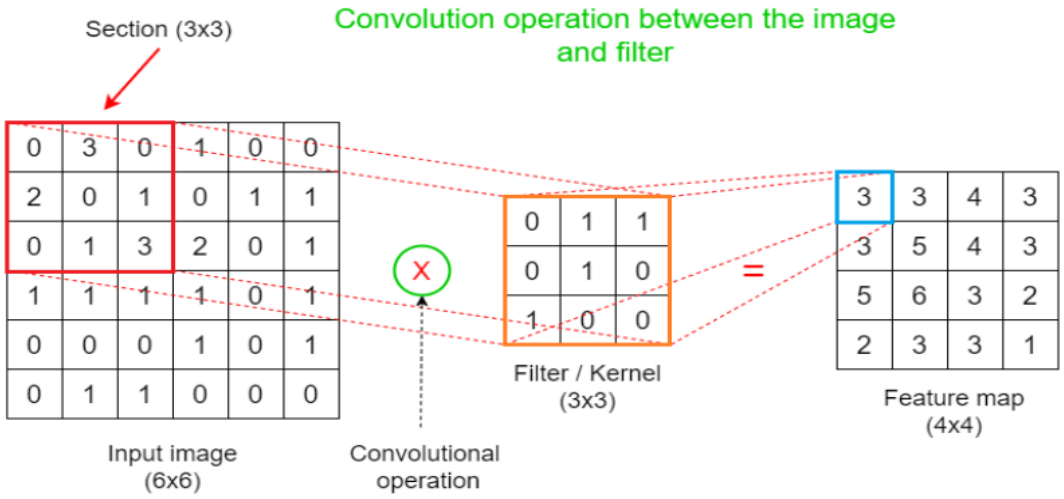
detects **vertical edges** in the image.

**Key Concepts:**

- **Filter Size**: Typically 3×3 or 5×5
- **Stride**: Step size for filter movement

- **Padding**: Adds extra borders to preserve size

**Application Question:**
Why are small filters with stride 1 helpful in camouflage detection?
**Answer:** Because they help capture even fine details such as cloth folds or rope edges that are crucial in spotting hidden objects.




- The filter performs element-wise multiplication and sums the result → produces a **feature map**.

**Convolution calculation**

The above diagram shows a convolution operation between an image section and a single filter. You can get row-wise or column-wise element multiplications and then summation.

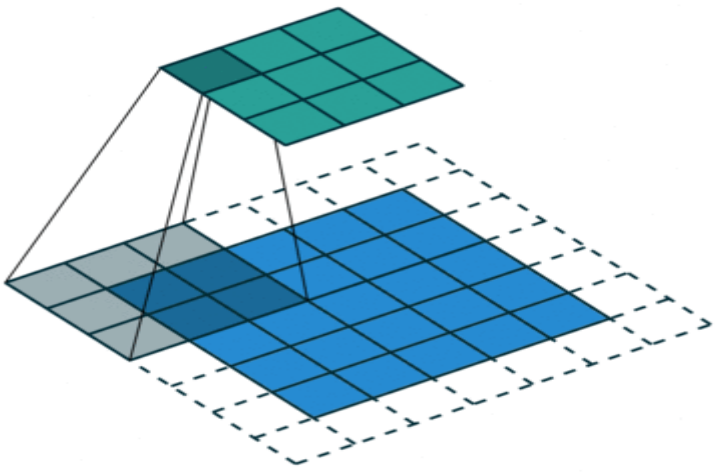
# Row-wise

(0*0 + 3*1 + 0*1) + (2*0 + 0*1 + 1*0) + (0*1 + 1*0 + 3*0) = 3

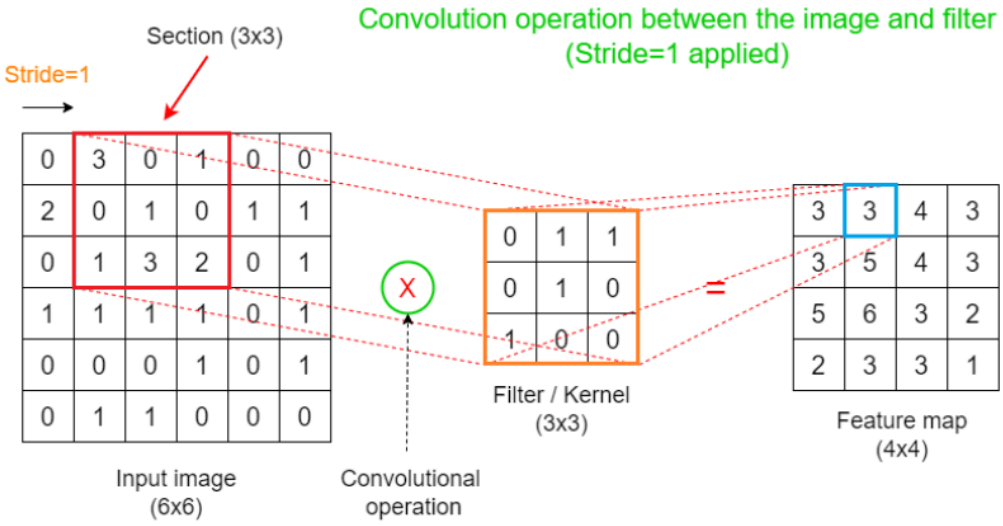
The result of this calculation is placed in the corresponding area in the feature map.

**Striding**

Some times we do not want to capture all the data or information available so we skip some neighboring cells let us visualize it,



Then we make another calculation by moving the filter on the image horizontally by one step to the right. The number of steps (pixels) that we shift the filter over the input image is called **Stride**. The shift can be done both horizontally and vertically. Here, we use Stride=1. Stride is also a hyperparameter that should be specified by the user.




Here also, we can get row-wise or column-wise element multiplications and then summation.

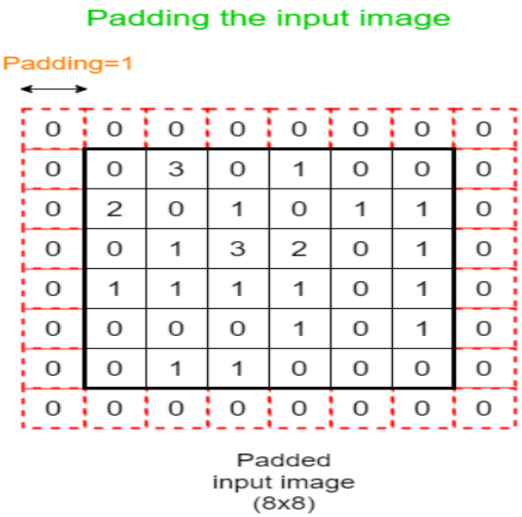
# Row-wise

(3*0 + 0*1 + 1*1) + (0*0 + 1*1 + 0*0) + (1*1 + 3*0 + 2*0) = 3

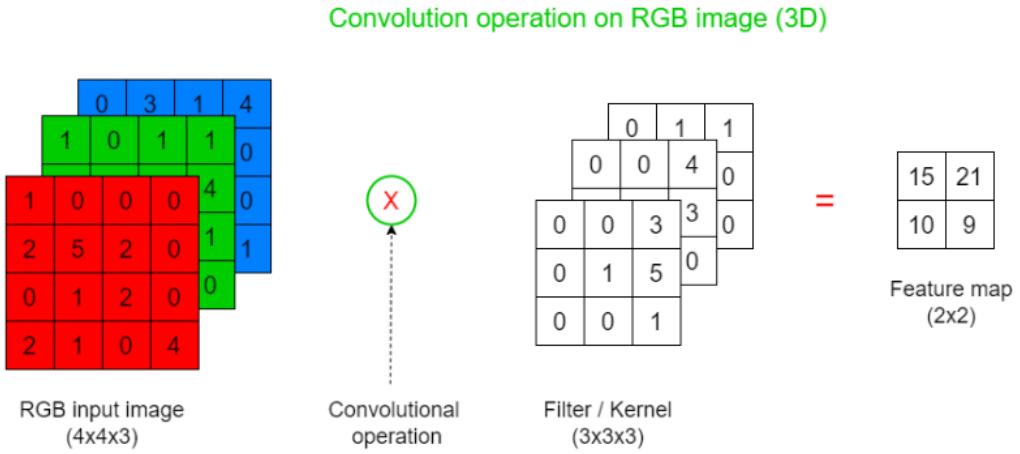
The result of this calculation is placed in the corresponding area in the feature map.

Likewise, we can do similar type of calculations by moving the filter on the image horizontally and vertically by one step (with Stride=1).

The size of the feature map is small than the size of the input image. The size of the feature map also depends on the Stride. If we use Stride=2, the size will be further reduced. If there are several convolutional layers in the CNN, the size of the feature map will be further reduced at the end so that we cannot do other operations on the feature map. To avoid this, we use apply **Padding** to the input image. Padding is a hyperparameter that we need to configure in the convolutional layer. It adds additional pixels with zero values to each side of the image. That helps to get the feature map of the same size as the input.

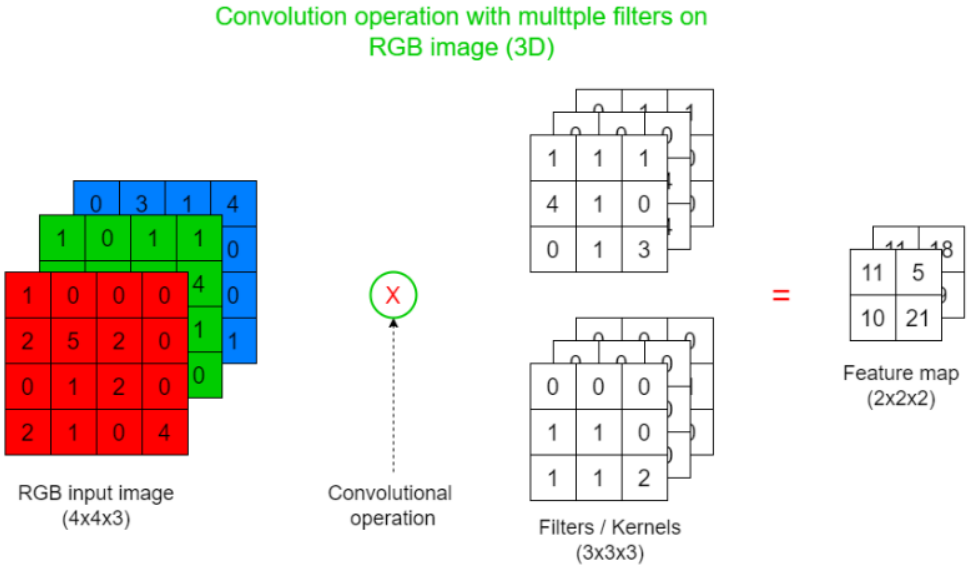



After applying Padding, the new size of the input image is (8, 8). If we do the convolution operation now with Stride=1, we get a feature map of size (6×6) that is equal to the size of the original image before applying Padding.




When the image is RGB, the filter should have 3 channels. This is because an RGB image has 3 color channels and 3-channel filters are needed to do the calculations.

Here, the calculation happens on each corresponding channel between the image section and the filter as previously. The final result is obtained by adding all outputs of each channel's calculations. That's why the feature map does not have a third dimension.




Here also, another dimension is added to the feature map. It is the third dimension which denotes the number of filters.

### 3. Pooling layers and pooling operation

Pooling layers are the second type of layer used in a CNN. There can be multiple pooling layers in a CNN. Each convolutional layer is followed by a pooling layer. So, convolution and pooling layers are used together as pairs.

**Objectives:**

- Extract the most important (relevant) features by getting the maximum number or averaging the numbers.
- Reduce the dimensionality (number of pixels) of the output returned from previous convolutional layers.
- Reduce the number of parameters in the network.
- Remove any noise present in the features extracted by previous convolutional layers.
- Increase the accuracy of CNNs.

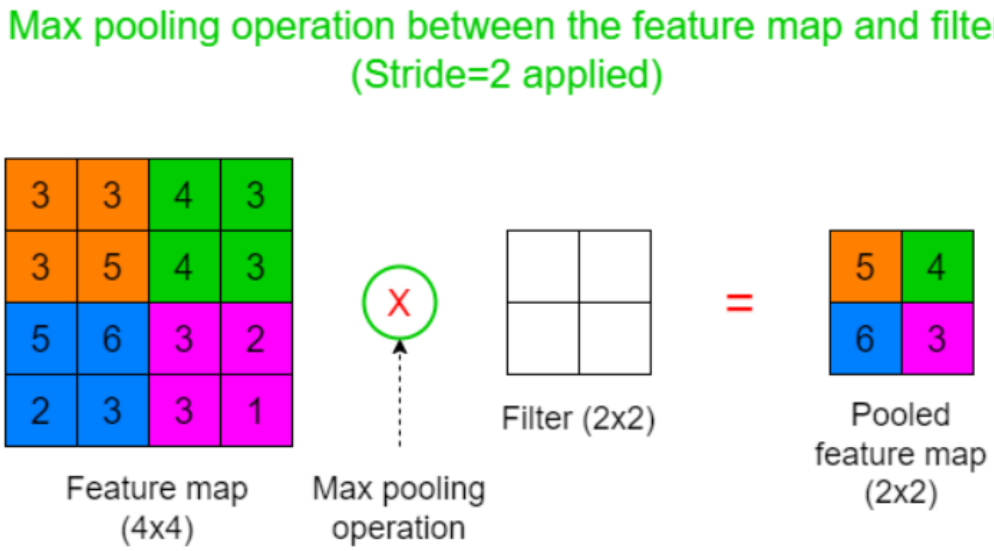
There are three elements in the pooling layer: **Feature map**, **Filter** and **Pooled feature map**. The ***pooling operation*** occurs in each pooling layer.

The pooling operation happens between a section of the feature map and the filter. It outputs the pooled feature map.

There are two types of pooling operations.

- **Max pooling:** Get the maximum value in the area where the filter is applied.
- **Average pooling:** Get the average of the values in the area where the filter is applied.

The following diagram shows the max pooling operation applied on the feature map obtained from the previous convolution operation.




**Filter:** This time, the filter is just a window as there are no elements inside it. So, there are no parameters to learn in the pooling layer. The filter is just used to specify a section in the feature map. The size of the filter should be specified by the user as a hyperparameter. The size should be smaller than the size of the feature map. If the feature map has multiple channels, we should use a filter with the same number of channels. The pooling operations will be done on each channel independently.

**Feature map section:** The size of the feature map section should be equal to the size of the filter we choose. We can move the filter vertically and horizontally on the feature map to create different sections. The number of sections depends on the Stride we use.

**Pooled feature map:** The pooled feature map stores the outputs of different pooling operations between different feature map sections and the filter. This will be the input for the next convolution layer (if any) or for the flatten operation.

**Stride and padding in the pooling operation**

- **Stride:** Here, the Stride is usually equal to the size of the filter. If the filter size is (2×2), we use Stride=2.
- **Padding:** Padding is applied to the feature map to adjust the size of the pooled feature map.

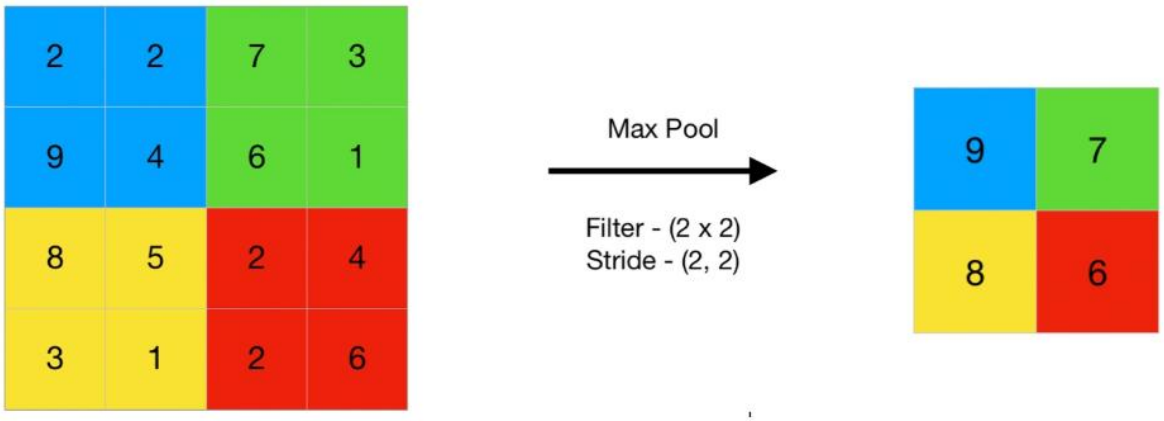
Both Stride and Padding are hyperparameters that we need to specify in the pooling layer.

**Note:** After applying pooling to the feature map, the number of channels is not changed. It means that we have the same number of channels in the feature map and the pooled feature map. If the feature map has multiple channels, we should use a filter with the same number of channels. The pooling operations will be done on each channel independently.

**Application** **Question:**

What happens if pooling is applied too aggressively in detecting small camouflaged objects?

**Answer:** It may discard small but important features like wires, straps, or patterns, hurting model performance.




## Summary Table

| Concept | Operation | Purpose | Effect on Size |
|---|---|---|---|
| Convolution | Filter slides over input | Detects patterns (edges, shapes) | Reduces (unless padded) |

| Concept | Operation | Purpose | Effect on Size |
|---|---|---|---|
| Stride | Filter step size | Controls resolution & speed | Higher stride = smaller output |
| Pooling | Downsamples feature map | Keeps strongest activations | Reduces |

### Why do we use convolution before pooling?

**→ To extract useful features before reducing them.**

### Why might a high stride hurt detection of small camouflaged objects?

**→ Because it skips over fine details and may miss thin features like ropes or mesh.**

### How does pooling improve generalization?

**→ It makes the network less sensitive to small shifts in the image (like camera angle).**

### 4. Activation Function (ReLU)

- After convolution, we apply ReLU:
  ReLU(x) = max(0, x)
- ReLU introduces non-linearity and keeps only positive activations.

**Application Question:**
Why is ReLU effective when detecting patterns like hidden metal rods among foliage?
**Answer:** It amplifies strong edges (like straight metal lines) and suppresses weak signals from noisy textures like leaves.

### 5. Convolution Blocks

- CNNs stack multiple layers of Conv → ReLU → Pooling
- Early layers detect **edges and shapes**
- Deeper layers detect **parts of objects** (like tires, weapons, backpacks)
- The output becomes more **abstract** and semantically meaningful

**Application Question:**
Why does stacking layers help CNNs detect complex patterns?

**Answer:** It lets the network build a **hierarchy of features** — from basic lines to full object structures like a tent flap or a rifle barrel.

### 6. Flatten Layer

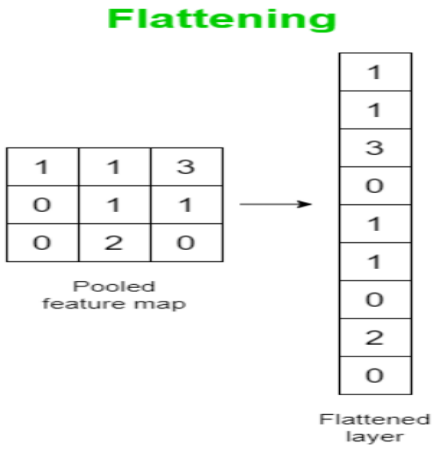
In a CNN, the output returned from the final pooling layer (i.e. the final pooled feature map) is fed to a Multilayer Perceptron (MLP) that can classify the final pooled feature map into a class label.

An MLP only accepts one-dimensional data. So, we need to flatten the final pooled feature map into a single column that holds the input data for the MLP.

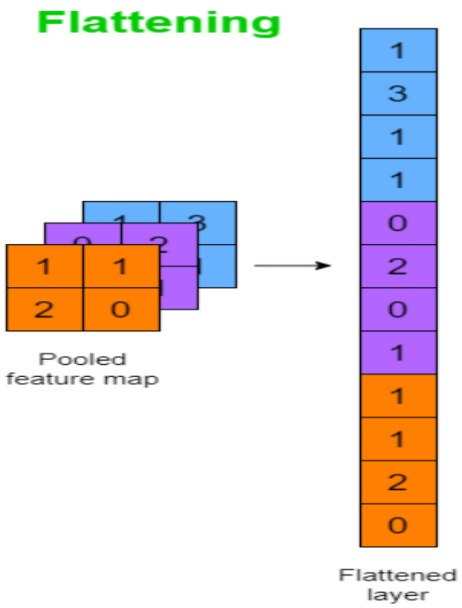
Unlike flattening the original image, important pixel dependencies are retained when pooled maps are flattened.

The following diagram shows how we can flatten a pooled feature map that contains only one channel.

**Application Question:**
What does flattening achieve in a CNN?
**Answer:** It turns spatial feature data into a vector so it can be used like a regular tabular input in dense layers.




The following diagram shows how we can flatten a pooled feature map that contains multiple channels.




## 7. Fully Connected (Dense) Layers

- Each node connects to all previous features.
- These layers **combine patterns** and perform the actual decision-making.

**Example:**

If previous filters detected a fabric fold + zip line + camouflage tone, one neuron may learn to recognize that as “tent.”

**Application Question:**

How do dense layers enable final predictions in camouflage detection?

**Answer:** By combining multiple high-level features into a final output, for example, if “zipper” + “tent fabric” + “green tone” are all present, it can classify as camouflage.

## 8. Output Layer (Softmax or Sigmoid)

- If you’re classifying:
    - **Binary**: Use **Sigmoid** → outputs a value between 0 and 1
    - **Multi-class**: Use **Softmax** → outputs probabilities for each class

**Example Output:**

[0.85, 0.10, 0.05] → High confidence it's camouflage.

**Question:**

Why is softmax ideal for multi-class outputs?

**Answer:** It ensures that probabilities sum to 1 and highlights the most likely prediction clearly.

**Visual Summary:**

| Layer | Function | Example |
|---|---|---|
| Input | Receives image | (224,224,3) |
| Conv | Extracts edges | 3×3 filters |
| ReLU | Keeps strong signals | max(0, x) |
| Pooling | Downsamples | 2×2 max pool |
| Conv Blocks | Learns shapes | multiple stacked |
| Flatten | Converts to 1D | e.g., 25088 |
| Dense | Combines features | 512 → 128 → 3 |
| Output | Predicts class | softmax |

## Why ReLU in Hidden Layers?

### What is ReLU?

ReLU (Rectified Linear Unit) is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

This means:

- If the input is positive, it stays the same
- If the input is negative, it becomes zero

**Why ReLU Is Preferred in Hidden Layers:**

| Feature | ReLU | Sigmoid | Tanh |
|---|---|---|---|
| Output Range | [0, ∞) | (0, 1) | (−1, 1) |

| Feature | ReLU | Sigmoid | Tanh |
|---|---|---|---|
| Derivative | 1 (for x>0), 0 (x≤0) | Very small for large inputs | Small for extreme inputs |
| Speed | Very fast | Slower | Slower |
| Saturation (Vanishing Gradients) | No | Yes | Yes |
| Introduces Non-Linearity | Yes | Yes | Yes |

### Understanding (Example)

**Use Case:** You're detecting camouflaged objects in a forest. Early CNN layers need to pick up **edges, contrast, curves, patterns**.

### Why ReLU works best here:

- It quickly **lets strong patterns through** (like sharp edges of a tent).
- It **blocks weak/noisy signals** (like leaf blur, natural clutter).
- It **doesn't squash large signals** like sigmoid/tanh do, so information is preserved.

**Question:**

Why does ReLU help more than sigmoid or tanh in detecting sharp contrast features in camouflage?

**Answer:**
Because it keeps the intensity of strong signals like vertical edges and object outlines intact, while suppressing weak signals. Sigmoid and tanh compress everything into a narrow range, making contrastless.

### Why Sigmoid in the Output Layer (Binary Classification)

### What is Sigmoid?

The sigmoid function is:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This maps any real value to a number between **0 and 1** — making it ideal for **binary probabilities**.

### Why Use Sigmoid in Output Layer:

| **Property** | **Sigmoid** |
|---|---|
| Output Range | (0, 1) — interpretable as probability |
| Application | Binary classification (e.g., camouflaged vs not) |
| Differentiable | Yes works with gradient descent |
| Final Decision | Usually thresholded at 0.5 |
| Works well with | Binary cross-entropy loss |

**Understanding (Example)**

**Use Case:** After processing image features, you want the CNN to output **"Is there a camouflaged object in this image?"** Yes or No.

### Why Sigmoid is best:

- It gives you a **probability score** like 0.82 or 0.03
- You can easily apply a threshold (e.g., > 0.5 = Yes)
- It works with binary labels like [0, 1]

**Question:**

Why is sigmoid more suitable than ReLU at the output layer for binary camouflage detection?

**Answer:**
Because we want a final value between 0 and 1 that can be interpreted as the **probability** of the presence of camouflage. ReLU can give values >1 or even 200, which aren't probabilities.

### Summary Table: ReLU vs Sigmoid

| **Criterion** | **ReLU (Hidden Layers)** | **Sigmoid (Output Layer)** |
|---|---|---|
| Range | 0 to ∞ | 0 to 1 |

| Criterion | ReLU (Hidden Layers) | Sigmoid (Output Layer) |
|---|---|---|
| Purpose | Keeps strong features alive | Converts final score to probability |
| Advantage | No vanishing gradients | Interpretable probability |
| Usage | Early feature extraction | Final binary classification |
| Example | Detecting edges, shapes | Detecting presence of object |

## 1. Softmax vs Sigmoid – When to Use What

- **Sigmoid**
  - Used in **binary classification**
  - Outputs a **single probability** (between 0 and 1)
  - Interpretation: Probability that the input belongs to the **positive class**

**Example:**

Camouflage Detection → 1: Camo, 0: No Camo
Sigmoid output: 0.87 → means 87% chance that camo exists

- **Softmax**
  - Used in **multi-class classification**
  - Outputs a **probability distribution** across multiple classes
  - Ensures all class probabilities **sum to 1**

**Example:**

Classify drone images into:

- Class 0: No camo
- Class 1: Tent camo
- Class 2: Vehicle camo

Softmax output:

csharp

CopyEdit

```
[0.05, 0.75, 0.20] → Highest score = Tent camo
```

### Key Differences:

| Feature | Sigmoid | Softmax |
|---|---|---|
| Problem Type | Binary | Multi-class |
| Output | Single probability | Vector of probabilities |
| Output Sum | No constraint | Sums to 1 |
| Use with | Binary Cross Entropy Loss | Categorical Cross Entropy Loss |
| Final Layer Neurons | 1 | Equal to number of classes |

**Question:**

Why is softmax preferred over sigmoid when detecting type of camouflage?

**Answer:**
Because softmax provides a probability for each class (e.g., tent camo, vehicle camo, or no camo), and ensures the total confidence is shared between them. Sigmoid would treat each class independently and might output inconsistent results.

### 2. Code Example Comparing ReLU and Sigmoid in Keras

Here's a simple example using both activations to show how they are used in different parts of the model.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Binary Classification Example
model_sigmoid = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(64, activation='relu'),      # Hidden layer with ReLU
    Dense(1, activation='sigmoid')     # Output layer with sigmoid
])
```

```
# Multi-Class Classification Example (3 classes)
model_softmax = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(64, activation='relu'),      # Hidden layer with ReLU
    Dense(3, activation='softmax')     # Output layer with softmax
])
```

## Applications of CNN

| Domain | Use |
|---|---|
| Healthcare | Brain tumor detection, X-ray analysis |
| Security | Face recognition, surveillance |
| Self-driving Cars | Lane detection, traffic sign recognition |
| Retail | Product image classification |
| Art | Style transfer (turn photo into painting style) |