# Convolutional Neural Network (CNN)

**Definition**: a type of deep learning model specifically designed for processing structured grid data, such as images. CNNs utilize convolutional layers to automatically learn spatial hierarchies of features from input images. They are particularly effective for tasks in computer vision, such as image classification, object detection, and image segmentation.

**Key Characteristics:**

* Convolutional Layers: Apply filters to the input to extract features.
* Pooling Layers: Reduce the dimensionality of feature maps, retaining important information while decreasing computational load.
* Fully Connected Layers: Connect every neuron in one layer to every neuron in the next layer, typically used at the end of the network for classification tasks.

## Convolution

**Definition -** a mathematical operation that combines two functions to produce a third function. In CNNs, it involves sliding a filter (kernel) over an input image to produce feature maps.

## Filters(Kernels)

**Definition**: small matrices used in convolution operations to detect specific features in an image, such as edges, textures, or patterns.

**Sobel Filter** – operatoris used in image processing and computer vision, particularly within edge detection algorithms where it creates an image emphasizing edges.

|  |  |
| --- | --- |
| Horizontal | Vertical |
|  |  |

**Scharr filter** - another method used to detect edges in digital images. It is an improvement over the Sobel operator.

|  |  |
| --- | --- |
| Horizontal | Vertical |
|  |  |

**Prewitt filter**

|  |  |
| --- | --- |
| Horizontal | Vertical |
|  |  |

## Padding

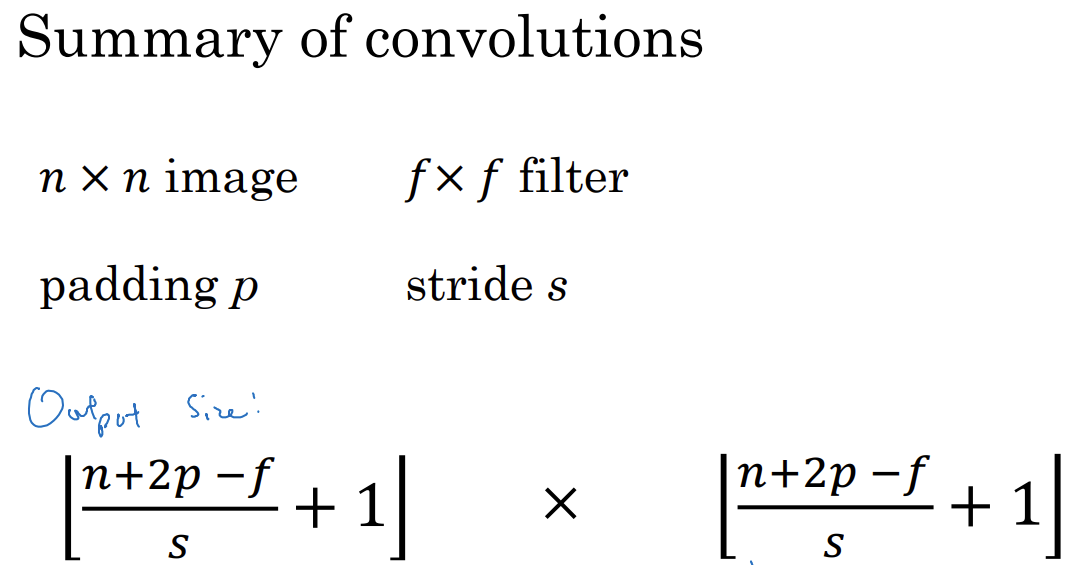
**Padding -** the process of adding extra pixels around the border of an image before applying a convolution operation. This helps preserve the spatial dimensions of the image and prevents loss of information at the edges.

**Padding types**:

* **Valid Padding**: No padding is added, which reduces the output size.
* **Same Padding**: Padding is added to ensure the output size is the same as the input size.

## Strided Convolutions

**Definition** : a technique where the filter moves across the image with a specified step size (stride). This reduces the spatial dimensions of the output feature map.



Output Size :

## Pooling Layers

**Definition**: Layers that reduce the spatial dimensions of feature maps, helping to decrease computation and control overfitting. Common types include max pooling and average pooling.

* **Max Pooling** – a pooling operation that selects the maximum value from a feature map within a defined window, effectively down sampling the input.
* **Average Pooling** - a pooling operation that calculates the average value from a feature map within a defined window, also down sampling the input.

## Why Convolutions?

Convolutions are essential in Convolutional Neural Networks (CNNs) for several reasons:

1. **Feature Extraction**:
   * Convolutions allow the model to automatically learn and extract important features from images, such as edges, textures, and shapes, without manual feature engineering.
2. **Spatial Hierarchy**:
   * They help capture spatial hierarchies in data. Lower layers can learn simple features (like edges), while deeper layers can learn more complex features (like objects).
3. **Parameter Sharing**:
   * Convolutions use the same filter (kernel) across the entire image, which reduces the number of parameters compared to fully connected layers. This makes the model more efficient and less prone to overfitting.
4. **Translation Invariance**:
   * Convolutions provide a degree of translation invariance, meaning that the model can recognize objects in different positions within the image.
5. **Reduced Computational Complexity**:
   * By using local connections and pooling, convolutions reduce the amount of computation required, allowing for faster training and inference.

# Variations of CNN

## A categorized overview

## 🔹 **1. Standard CNNs**

These include early architectures like:

* **LeNet-5**: One of the first CNNs, used for digit recognition.
* **AlexNet**: Popularized deep learning for image classification; introduced ReLU and dropout.
* **VGGNet**: Used only 3×3 convolutions; very deep but with a simple, uniform structure.

### 🔹 2. Modular Architectures

These CNNs use building blocks or modules instead of simple stacking of layers:

**✅ GoogLeNet / Inception**

* Introduced **Inception modules**: multiple convolution filters (1×1, 3×3, 5×5) and pooling in parallel.
* Used **1×1 convolutions** for dimensionality reduction.

**✅ ResNet (Residual Networks)**

* Introduced **skip connections** (residual connections) to combat vanishing gradients.
* Enabled training of **very deep networks** (e.g., 50, 101, or 152 layers).

**✅ DenseNet**

* Each layer receives input from **all previous layers** (dense connectivity).
* Helps with **gradient flow** and feature reuse.

### 🔹 3. Efficiency-Oriented CNNs

Designed for mobile devices and low-resource settings:

**✅ MobileNet**

* Uses **Depthwise Separable Convolutions** to reduce computation.
* Suitable for mobile and embedded vision applications.

**✅ ShuffleNet**

* Uses **pointwise group convolutions** and **channel shuffling** for speed and efficiency.

**✅ SqueezeNet**

* Very small model with **Fire modules** (1×1 and 3×3 convolutions) that achieves AlexNet-level accuracy with fewer parameters.

### 🔹 4. Dilated/Deformable CNNs

For capturing wider context or more flexible spatial features:

**✅ Dilated Convolutions (Atrous)**

* Adds **holes/spaces** in the kernel to **increase receptive field** without increasing parameters.
* Useful in semantic segmentation (e.g., DeepLab).

**✅ Deformable Convolutions**

* The kernel learns **offsets** dynamically to adapt to object shapes and positions.
* Enhances object detection and segmentation accuracy.

### 🔹 5. Capsule Networks (CapsNet)

Proposed by Geoffrey Hinton.

* Maintains **spatial relationships** between features using capsules.
* Uses **dynamic routing** instead of pooling.

### 🔹 6. Attention-Augmented CNNs

* Integrate **attention mechanisms** (like in Transformers) into CNNs to help the model focus on relevant regions.
* Example: **CBAM (Convolutional Block Attention Module)** or **Squeeze-and-Excitation Networks**.

### 🔹 7. Hybrid CNN-Transformer Architectures

* Combine **the local feature learning** of CNNs with **the global modeling** of Transformers.
* Examples: **ConvNeXt, CoAtNet, ViT + CNN hybrids**.

### 🔹 Summary Table

|  |  |  |
| --- | --- | --- |
| Type | Key Feature | Example |
| Standard | Stacked conv/pool/FC layers | LeNet, VGG, AlexNet |
| Modular | Blocks with diverse paths or skips | Inception, ResNet |
| Efficient | Lightweight for mobile | MobileNet, SqueezeNet |
| Context-aware | Enlarged receptive field | Dilated CNNs |
| Spatially flexible | Adapts to object shape/position | Deformable CNNs |
| Relationship-aware | Encodes part-whole spatial hierarchy | Capsule Networks |
| Attention-based | Highlights important features | SE-Net, CBAM |
| Hybrid | Combines CNNs with Transformers | ConvNeXt, CoAtNet |

**More info**

### 🔹 1. LeNet-5 (1998)

 **Architecture**: 2 CONV layers → 2 Pooling → FC layers

 **Use Case**: Handwritten digit recognition (MNIST)

 **Key Features**:

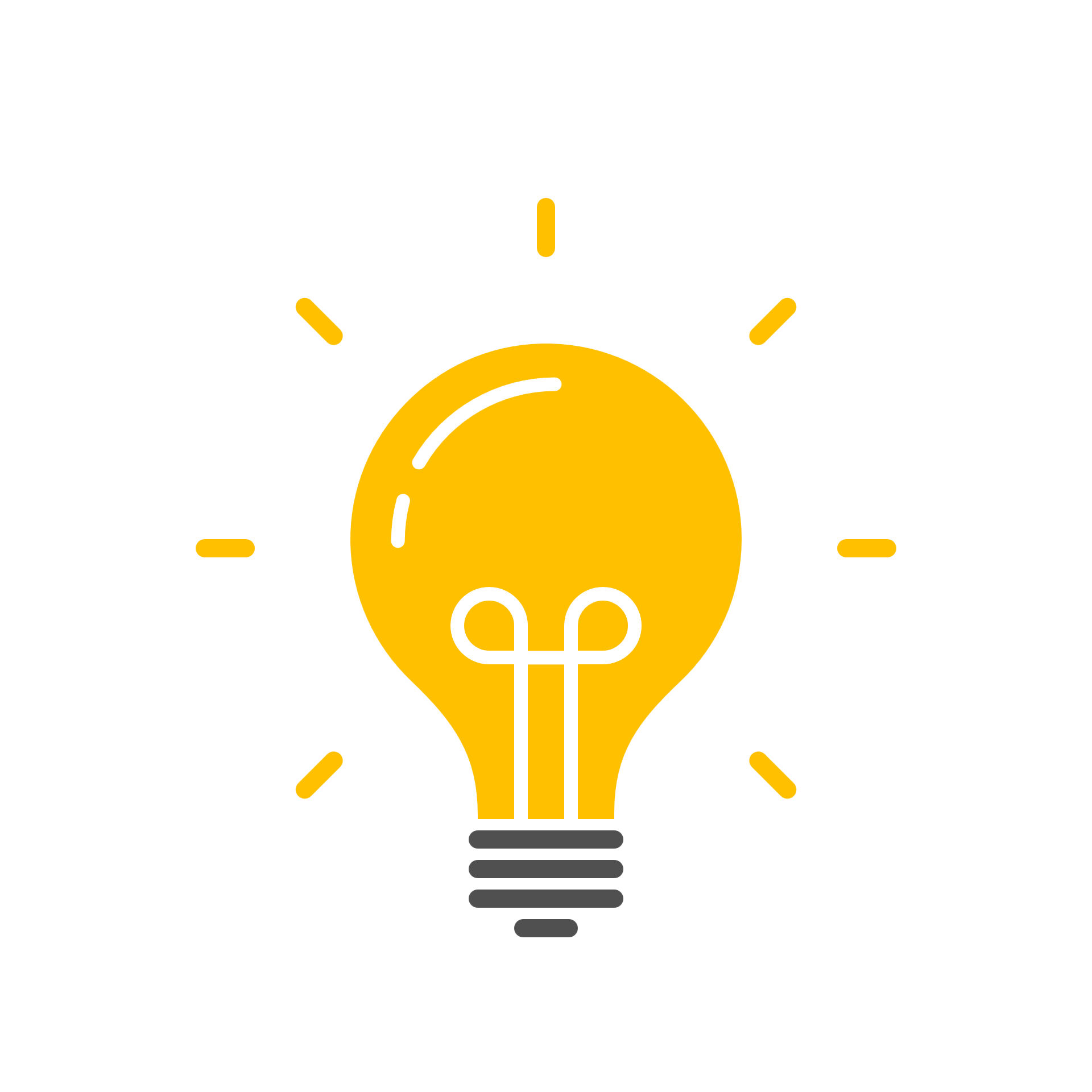
* No padding: spatial dimensions reduce after conv.
* Sigmoid/tanh activations.
* Introduced the idea of local receptive fields and weight sharing.
* Limitation: Too shallow for complex tasks**.**

**🔹** 2. AlexNet (2012)

* **Architecture**: 5 CONV layers + 3 FC layers
* **Use Case**: ImageNet classification
* **Key Innovations**:
  + **ReLU activation** (much faster training).
  + **Dropout** in FC layers to reduce overfitting.
  + GPU training with parallelization.
  + Local response normalization (LRN, now deprecated in modern nets).

### 🔹 3. VGGNet (2014)

* **Architecture**: Deep stack of 3×3 convolutions + max pooling + FC
* **Key Idea**: Depth improves performance.
* **Pros**:
  + Very simple and uniform design (repeating blocks).
  + Excellent baseline for transfer learning.
* **Cons**:
* Heavy model (millions of parameters).
* Computationally expensive.

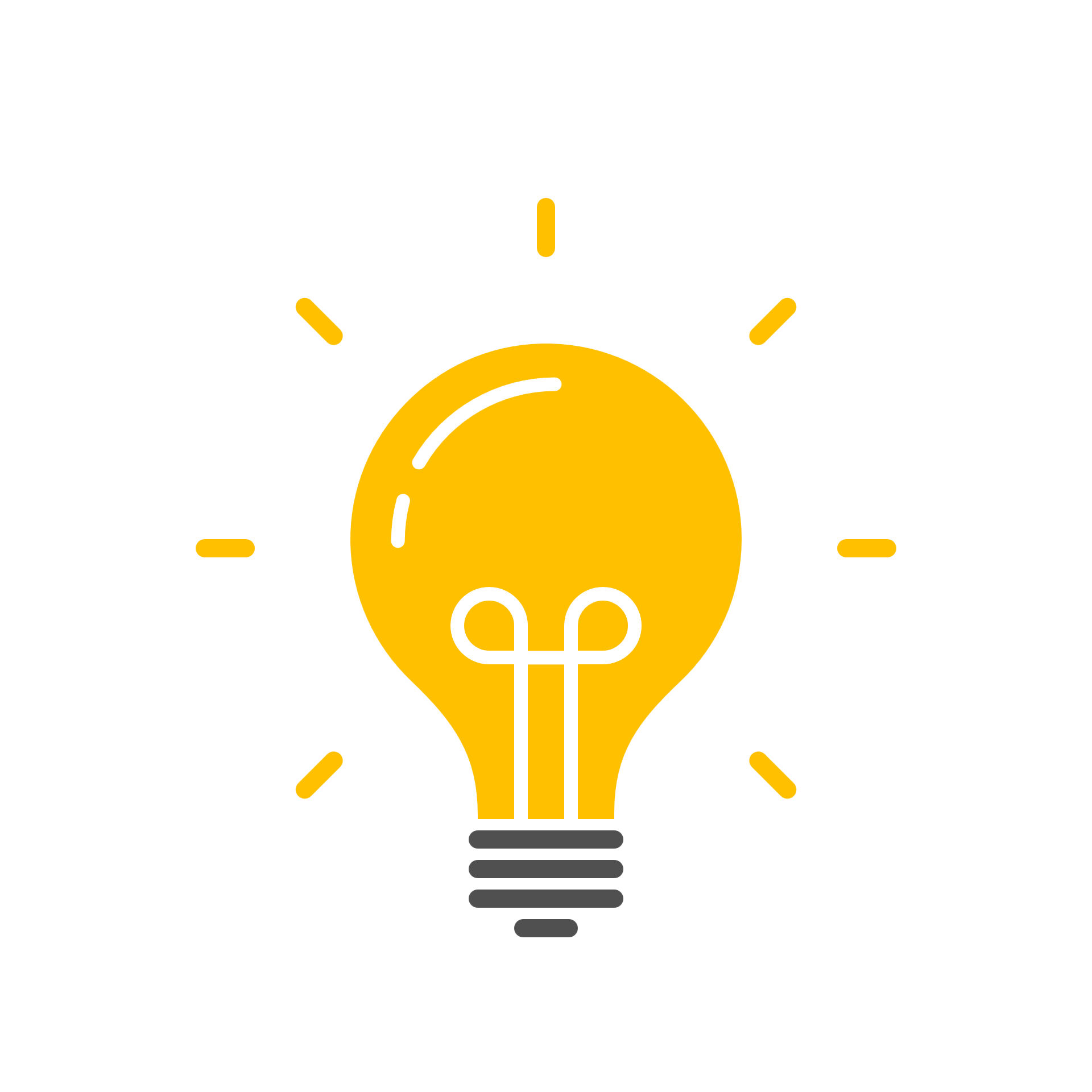
***Next networks move away from plain sequential stacks and use modular blocks to improve learning and gradient flow.***

**🔹 4. GoogLeNet (Inception Network) (2014)**

* **Inception Module**: Parallel branches of:
  + 1×1 conv (bottleneck)
  + 3×3 conv
  + 5×5 conv
  + Max pooling
* **Key Concepts**:
  + **1×1 convolutions** for dimensionality reduction.
  + **Factorization** of large convolutions into smaller ones (e.g., 5×5 → two 3×3).
  + Efficient and deep with fewer parameters than VGG.
* Very deep, but computationally efficient.

**🔹 5.** ResNet (2015)

* **Residual blocks:** Output = F(x) + x (skip connections)
* **Motivation:** Solve vanishing gradient and degradation problem in deep nets.
* **Impact:**
  + Enabled networks with **100+ layers.**
  + Greatly improved training of very deep architectures.

***Next networks *focused on reducing model size, computations, and inference time, especially for edge devices.**

**🔹** 6. MobileNet (2017)

* **Core Idea: Depthwise Separable Convolutions**
  + **Depthwise:** One filter per input channel.
  + **Pointwise (1×1):** Combines outputs across channels.
* **Benefit:** Reduces parameters and FLOPs by up to 9x.
* **Use Case**: Mobile vision apps, embedded AI.

**🔹** 7. ShuffleNet (2018)

* **Key Techniques:**
  + **Grouped convolutions:** Split channels into groups.
  + **Channel shuffle:** Mix info across groups.
* **Goal:** Maintain accuracy while **further reducing cost** beyond MobileNet**.**

**🔄 Quick Comparison**

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | LeNet / AlexNet / VGG | Inception / ResNet | MobileNet / ShuffleNet |
| Structure | Sequential | Modular blocks | Lightweight modules |
| Depth | Low–Medium | Deep (e.g., 50+ layers) | Shallow/Compact |
| Parameters | High (VGG) | Optimized via modules | Very low |
| Speed | Slow (VGG) | Faster | Fast (real-time) |
| Use Cases | Vision classification | Classification + Detection | Edge/mobile inference |

# Q&A

**✅** What Batch Normalization Does:

For each feature channel c, it:

1. **Computes mean and variance**:



1. **Normalizes**:



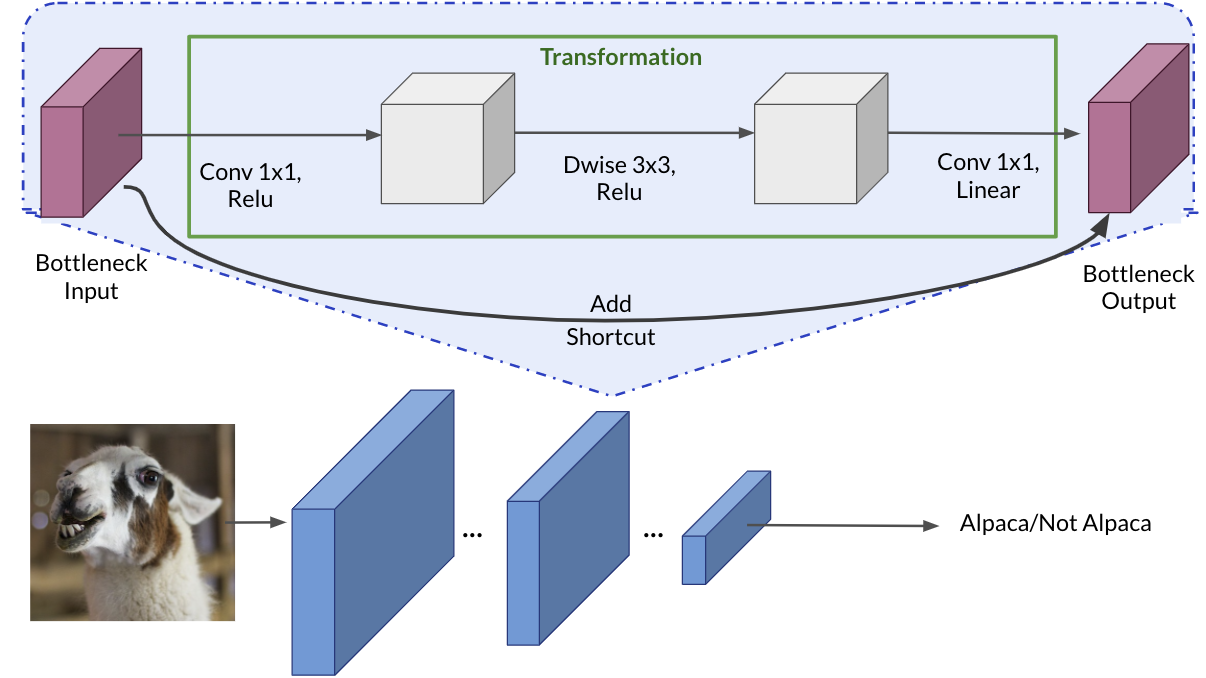
1. **Scales and shifts** (learnable parameters):



**🧠 Why Use It?**

* Stabilizes training by normalizing activations.
* Helps with faster convergence.
* Reduces sensitivity to weight initialization.
* Acts as a mild regularizer.

**✅** MobileNetV2 bottleneck block

****

**🔍** Why ReLU after the first 1x1 convolution?

This first **1x1 convolution** is used to **expand the number of channels** — it's often called the **expansion layer**.

* It increases the representational power of the network.
* Applying **ReLU** introduces **non-linearity**, which is essential for the network to learn complex patterns.
* Without non-linear activation, stacking layers would just be a linear function of the input — not helpful.

So:  
**👉 ReLU here adds learning capacity** after expanding the channel space**.**

**🔍** Why Linear activation after the last 1x1 convolution?

The final **1x1 convolution** is a **projection layer** — it reduces the number of channels back to the original input size. This is key for:

* **Maintaining dimensional consistency** for the shortcut connection (residual addition).
* Ensuring the **residual connection adds cleanly** without destroying information.

🧠 If we apply **ReLU here**, we risk **information loss**:

* ReLU zeroes out all negative values — that can block gradient flow or suppress important signals.
* We want to **preserve the full range** of learned features before adding them to the shortcut.

So:  
👉 **Linear activation here helps preserve information** and makes the residual connection more effective.

**🔁 Summary:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Layer** | **Purpose** | **Activation** | **Why?** |
| **Conv 1x1 (expand)** | Increase channels, more features | ✅ ReLU | Learn non-linear patterns |
| **Dwise 3x3** | Efficient spatial filtering | ✅ ReLU | Add non-linearity |
| **Conv 1x1 (project)** | Reduce channels, restore input shape | 🚫 Linear only | Preserve information for residual addition |

### Data Augmentation

#### **PCA color augmentation**

**PCA color augmentation** (also known as **PCA lighting** or **AlexNet-style lighting noise**) is a data augmentation technique applied to RGB images to introduce color variation in a way that mimics natural changes in lighting conditions. It was introduced in the **AlexNet** paper by Krizhevsky et al. (2012) to help improve the generalization of convolutional neural networks.

**🔍 Concept**

Principal Component Analysis (PCA) is applied to the RGB color values of all pixels in the training dataset to find the **major axes of color variation**. Then, at training time, small perturbations are added along these principal components to simulate lighting variations.

**Why use PCA color augmentation?**

* To **increase robustness** to lighting conditions and color variations.
* Helps prevent overfitting by making the model less sensitive to the specific color distribution in the training set.
* Mimics realistic changes like shadows, illumination shifts, or camera white balance effects.

**🧠 How it works (step-by-step)**

1. **Flatten the image dataset**:
   * Stack all RGB pixel values from all images into an N×3N \times 3N×3 matrix, where NNN is the total number of pixels across the dataset and 3 corresponds to R, G, B channels.
2. **Compute the covariance matrix** of the RGB values.
3. **Perform PCA**:
   * Get eigenvectors (principal components) and eigenvalues (variance explained).
4. **Perturb the input image**:
   * At runtime, for each image, generate a noise vector α\alphaα of random values drawn from a Gaussian distribution (e.g., N(0,0.1)
   * Compute the color shift:

ΔI=P⋅(α⋅λ)

where:

* + - P: matrix of principal components (3×3)
    - λ: eigenvalues
    - α: random scalar coefficients

1. **Add the color shift** to each RGB pixel in the image.

💡 **Example (PyTorch-style pseudocode)**

def pca\_color\_augmentation(image, eigvecs, eigvals, alpha\_std=0.1):

alpha = np.random.normal(0, alpha\_std, size=(3,))

rgb\_shift = np.dot(eigvecs, eigvals \* alpha)

return np.clip(image + rgb\_shift, 0, 255)

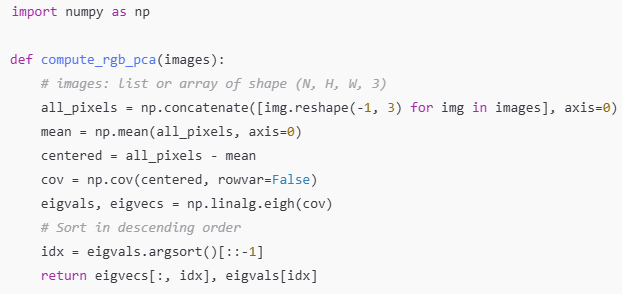
**⚠️ Things to watch out for**

* PCA lighting augmentation can be subtle and shouldn't drastically alter colors — it's meant to **simulate lighting**, not change image semantics.
* It’s **less commonly used** today than other augmentations (like random color jitter) but can be beneficial for certain vision tasks.

### ✅ ****Step-by-Step PCA Color Augmentation in TensorFlow****

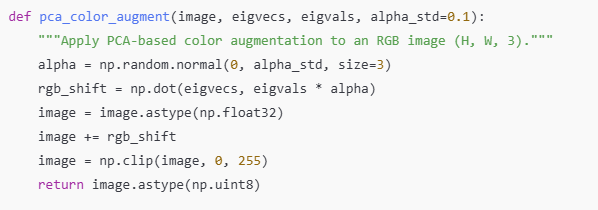
#### 1. **Precompute PCA on RGB pixel values**

Before training, run this **once on your dataset**:



Save eigvecs and eigvals for use during augmentation.

#### 2. PCA Lighting Function



#### 3. Wrap it for use in a TensorFlow pipeline



#### 4. Use in a Dataset Pipeline



**📌 Notes**

* PCA color augmentation is **more computationally expensive** than basic color jittering, so use it when color robustness is critical.
* You can replace .astype(np.uint8) and normalization steps based on your model’s input format.