Course Module: Convolutional Neural Networks

Latreche Sara

Contents

1 1. Motivation & Intuition

Note

Traditional fully connected networks treat every pixel equally, which is inefficient for images. CNNs solve this by leveraging spatial structure.

Two key insights:

- **Locality:** relevant features (edges, textures) are local in space.
- **Translation invariance:** the same feature can appear anywhere.

Applications: computer vision, speech, medical imaging, etc.

2 Historical Roots of Convolutional Neural Networks

Historical and Neuroscience Perspective

CNNs are not only mathematical constructs but also the result of decades of research in neuroscience and computer vision. They are part of humanity's long-standing attempt to understand **how the brain sees**.

Neuroscience Foundations (1960s)

In the 1960s, **David Hubel and Torsten Wiesel** showed that neurons in the visual cortex respond selectively to edges and orientations. They discovered:

- **Simple cells:** activated by specific edge orientations in fixed positions.
- **Complex cells:** activated by edges regardless of exact position.

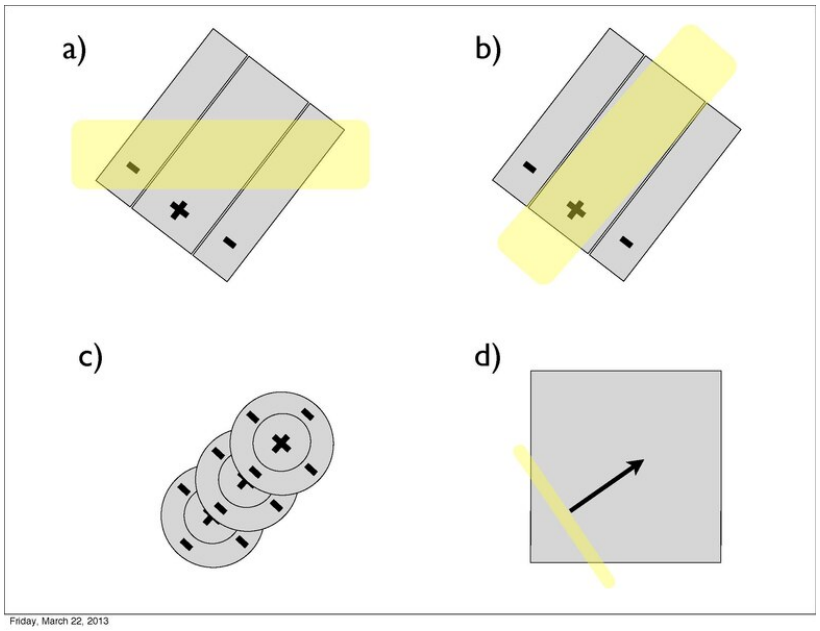


Figure 1: Hierarchical organization of the visual cortex: simple cells (edges) feed into complex cells (invariance). Source: Wikimedia Commons, CC-BY-SA.

Hubel & Wiesel's work earned the **1981 Nobel Prize in Physiology or Medicine** and still inspires CNN design today.

—

Hand-crafted Filters (1970s–1980s)

Inspired by these biological insights, researchers created filters that mimic edge detection: - **Sobel operator (1968)** - **Prewitt operator (1970s)** - **Gabor filters (1980s)**

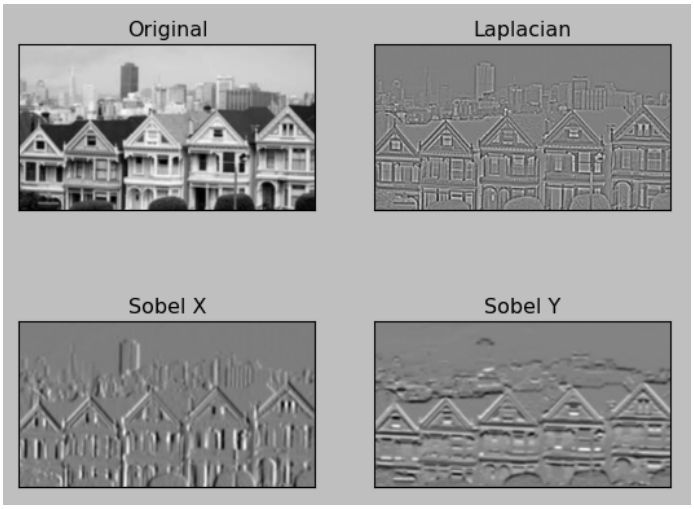


Figure 2: Sobel operator applied to an image: left = original, right = edge map.

—

Birth of CNNs (1980–1998)

Kunihiko Fukushima proposed the *Neocognitron* (1980), introducing: - Local receptive fields, - Weight sharing, - Subsampling (early pooling).

Yann LeCun later implemented these ideas in *LeNet-5* (1998), used for handwritten digit recognition (MNIST).

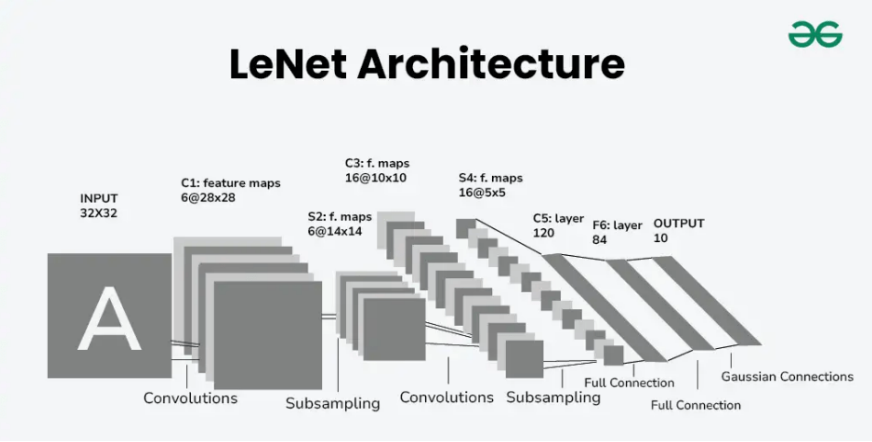


Figure 3: Architecture of LeNet-5, one of the first CNNs successfully applied to handwritten digit recognition. Source: LeCun et al. (1998) Link

—

The Deep Learning Revolution (2012)

CNNs became mainstream after the success of **AlexNet** (Krizhevsky et al., 2012), which won the ImageNet competition by a huge margin.

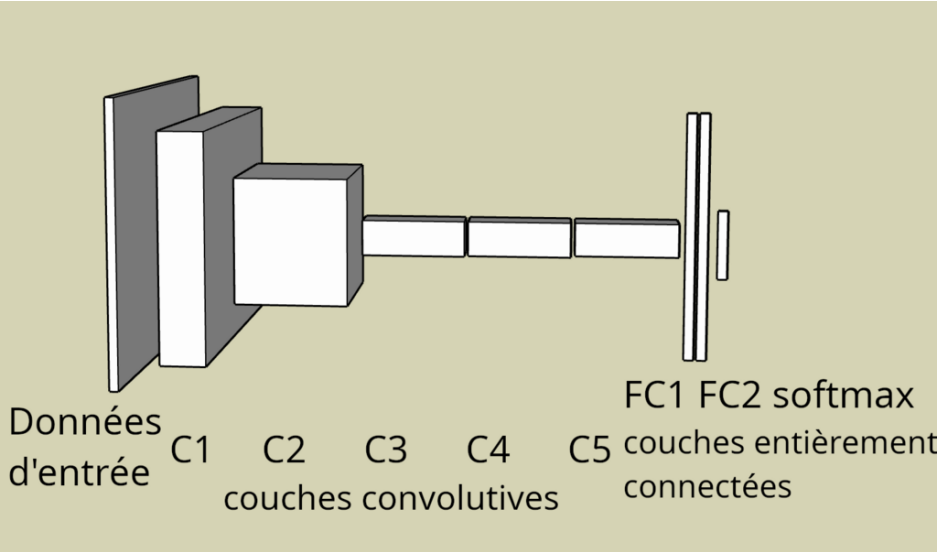


Figure 4: AlexNet architecture, which revolutionized computer vision in 2012. Source: Krizhevsky et al. (2012), ImageNet paper. Link

—

Modern CNN Architectures (2015–Today)

- **VGGNet (2014):** small 3×3 kernels, deep networks. - **ResNet (2015):** skip connections, 152+ layers. - **MobileNet/EfficientNet (2017+):** efficient CNNs for mobile and edge devices.

—

Timeline of CNN History

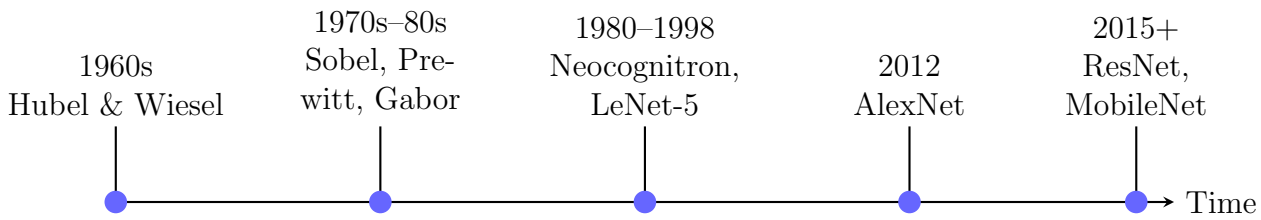


Figure 5: Timeline of CNN development, from neuroscience to deep learning.

3 Image Filters and Convolutions

Note

An **image filter** is a function that scans a local neighborhood of pixels and detects whether a certain pattern (such as an edge or a corner) is present in that region.

1D Example: Binary Image

To build intuition, let us start with a simple case: a **1-dimensional binary image** (a row of black/white pixels) and a filter F of size two.

The filter is a short vector of weights that we slide across the image. At each position, we compute the dot product between the filter and the corresponding local patch of pixels.

Definition

If X is the input image of length d, the output image Y is given by:

$$Y_i = F \cdot (X_{i-1}, X_i)$$

To ensure the output has the same size as the input, we usually add zeros at the borders of X. This is called **padding**.

Concrete Example

Let the filter be $F_1 = (-1, +1)$. When we convolve it with the 1D binary image shown below, it produces a new sequence that highlights positions where a transition from 0 to 1 occurs.

Example

Input image (1D binary pixels):

$$X = [0,\ 0,\ 1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0]$$

Filter:

$$F_1 = [-1,\ +1]$$

Output after convolution:

$$Y = [0,\ 1,\ 0,\ 0,\ -1,\ 1,\ -1,\ 0,\ 0]$$

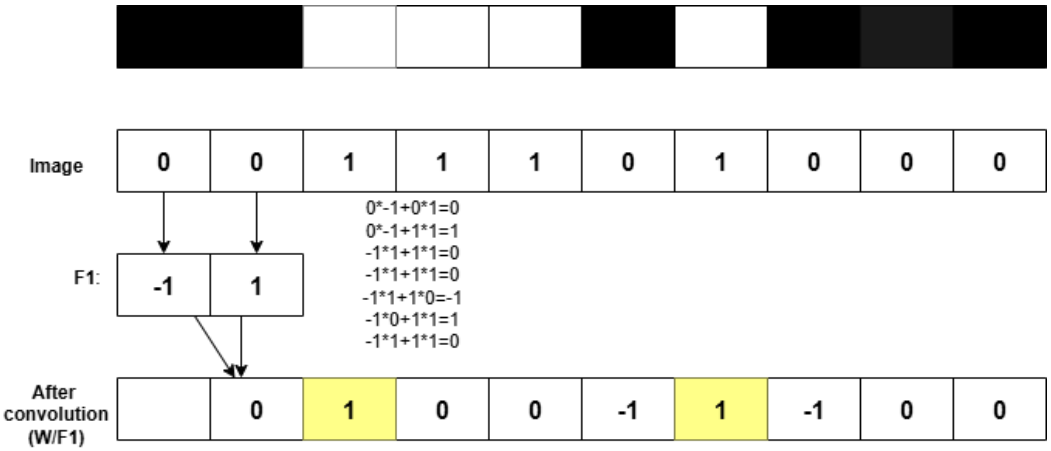


Figure 6: Applying filter $F_1 = [-1, +1]$ to a 1D binary image. The highlighted positive responses correspond to left edges in the image (transitions $0 \rightarrow 1$).

Another interesting filter is $F_2 = (-1, +1, -1)$. When applied to the same binary image, F_2 responds strongly to **isolated bright pixels**, i.e., when a '1' is surrounded by '0's.

Second Example: Isolated Pixel Detection

Now let us try a different filter:

$$F_2 = [-1,\ +1,\ -1]$$

This filter examines three consecutive pixels at a time. It responds strongly only when the central pixel is bright (1) and both its neighbors are dark (0). In other words, it acts as a **detector for isolated positive pixels**.

Example

Input image:

$$X = [0,\ 0,\ 1,\ 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0]$$

Filter:

$$F_2 = [-1,\ +1,\ -1]$$

Output after convolution:

$$Y = [-1,\ 0,\ -1,\ 0,\ -2,\ 1,\ -1,\ 0]$$

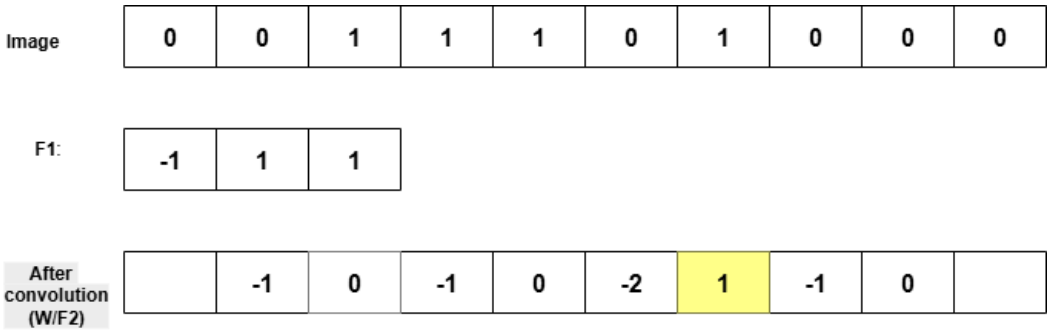


Figure 7: Applying filter $F_2 = [-1, +1, -1]$ to a 1D binary image. Notice how the positive response (+1, highlighted) occurs when the central pixel is isolated (a 1 surrounded by 0s).

Study Question

Why does F_2 ignore regions with multiple consecutive 1s but respond to a single isolated 1?

4 Edges and Edge-Detection Filters

Definition

Edge: In image processing, an edge is a region in the image where the intensity (or color) changes sharply. Edges correspond to object boundaries, texture changes, or lighting transitions, and they are fundamental features used by CNNs to build higher-level representations.

Directional Edge Filters

Just as in our 1D examples, filters in 2D can be designed to detect intensity changes in specific directions:

- **Left Edge Filter:** Detects transitions from dark to bright along the horizontal axis.

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

- **Right Edge Filter:** Detects transitions from bright to dark along the horizontal axis.

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

- **Top Edge Filter:** Detects transitions from dark to bright vertically.

$$K = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

- **Bottom Edge Filter:** Detects transitions from bright to dark vertically.

$$K = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

Prewitt Operator

The **Prewitt operator** uses 3×3 kernels that approximate the derivative in horizontal and vertical directions.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad K_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

- K_x: detects vertical edges (left vs. right). - K_y: detects horizontal edges (top vs. bottom).

Note

Variants such as Prewitt-Top, Prewitt-Bottom, Prewitt-Left, and Prewitt-Right emphasize edges in specific directions by modifying the weight distribution.

Sobel Operator

The **Sobel operator** is similar to Prewitt but gives more weight to the center row or column, making it more robust to noise.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- K_x: detects vertical edges. - K_y: detects horizontal edges.

Key Insight

Edge filters are the building blocks of convolutional neural networks. - Early CNN layers often learn filters resembling Prewitt or Sobel operators. - Higher layers build on these edges to detect shapes, textures, and objects.

Think About It

Why would a CNN prefer to learn edge filters automatically instead of using fixed ones like Sobel or Prewitt?

Convolution vs. Correlation

In classical signal processing, convolution and correlation are distinct operations. In deep learning, however, most libraries implement the correlation operation but still refer to it as **convolution**.

Note

This naming difference does not affect learning: CNNs can discover the correct filters regardless of whether the operation is technically convolution or correlation.

Concrete Example

Let the filter be $F_1 = (-1, +1)$. Sliding this filter across a binary 1D image highlights positions where a transition from 0 to 1 occurs — effectively acting as a **left-edge detector**.

Example

Input image (binary row of pixels):

$$X = [0, 0, 1, 1, 1, 0, 1, 0, 0, 0]$$

Filtered output:

$$Y = [\,0,\ 1,\ 0,\ 0,\ -1,\ 1,\ -1,\ 0,\ 0\,]$$

Another filter, $F_2 = (-1, +1, -1)$, responds strongly to isolated bright pixels (detecting small "dots" in the binary image).

Interpretation

- Filters act as **pattern detectors**. - Different filters highlight different aspects of the image (edges, dots, textures). - The result of applying a filter is called a **feature map**.

Hands-on Lab

Hands-on: Implement a 1D convolution in Python.

```
import numpy as np

# Input 1D binary "image"
X = np.array([0,0,1,1,1,0,1,0,0,0])

# Define filter
F = np.array([-1, 1])

# Convolve manually
Y = []
for i in range(1, len(X)):
    patch = X[i-1:i+1]   # local neighborhood
    Y.append(np.dot(F, patch))

print("Output:", Y)
```

Try replacing F with '[-1, 1, -1]' and observe the result.

Hands-on Lab

Implement a 3×3 filter in NumPy and apply it to a grayscale image. Visualize edge detection.

5 Filter Banks and Tensor Representations

Note

Two-dimensional filters similar to those we have seen are thought to exist in the **visual cortex** of all mammalian brains. Statistical analysis of natural images also shows that edge-like patterns occur frequently, which may explain why such filters are biologically relevant.

From Filter Banks to Channels

Early computer vision researchers often designed **filter banks** by hand. A **filter bank** is simply a collection of filters applied to the same input image.

Definition

If k filters are applied to an image of size $n \times n$, the result is a stack of k new images, called **channels**, producing an output tensor of shape $n \times n \times k$.

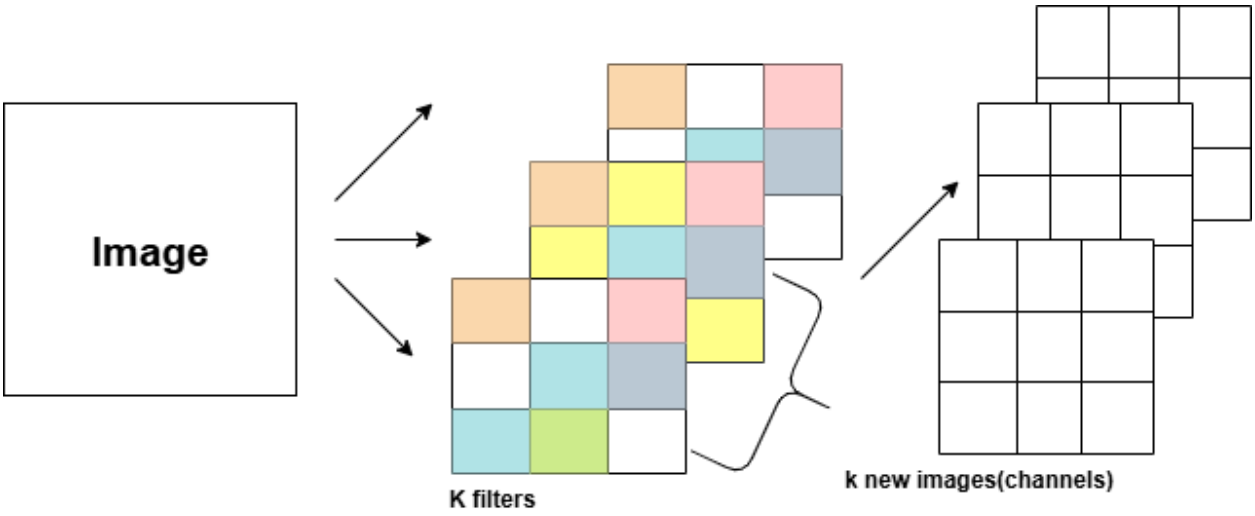


Figure 8: Illustration of a filter bank: multiple filters applied to one input image produce multiple channels, which can be stacked into a 3D tensor. imagine stacking all these new images up so that we have a cube of data, indexed by the original row and column indices of the image, as well as by the channel. The next set of filters in the filter bank will generally be three-dimensional: each one will be applied to a sub-range of the row and column indices of the image and to all of the channels.

Tensors in CNNs

Once we stack multiple channels, we obtain a **tensor** representation of the data. Subsequent filters must therefore be three-dimensional: they span both the *spatial dimensions* (row and column indices) and the *channel dimension*.

Example

Suppose we have two 3×3 filters in the first layer:

$$f_1 = (\text{vertical edge detector}), \quad f_2 = (\text{horizontal edge detector}).$$

- Input: an $n \times n$ grayscale image. - Output: an $n \times n \times 2$ tensor (two channels: vertical and horizontal edges). - Next: a 3D filter can look for patterns like "two horizontal bars + two vertical bars," producing a single final $n \times n$ feature map.

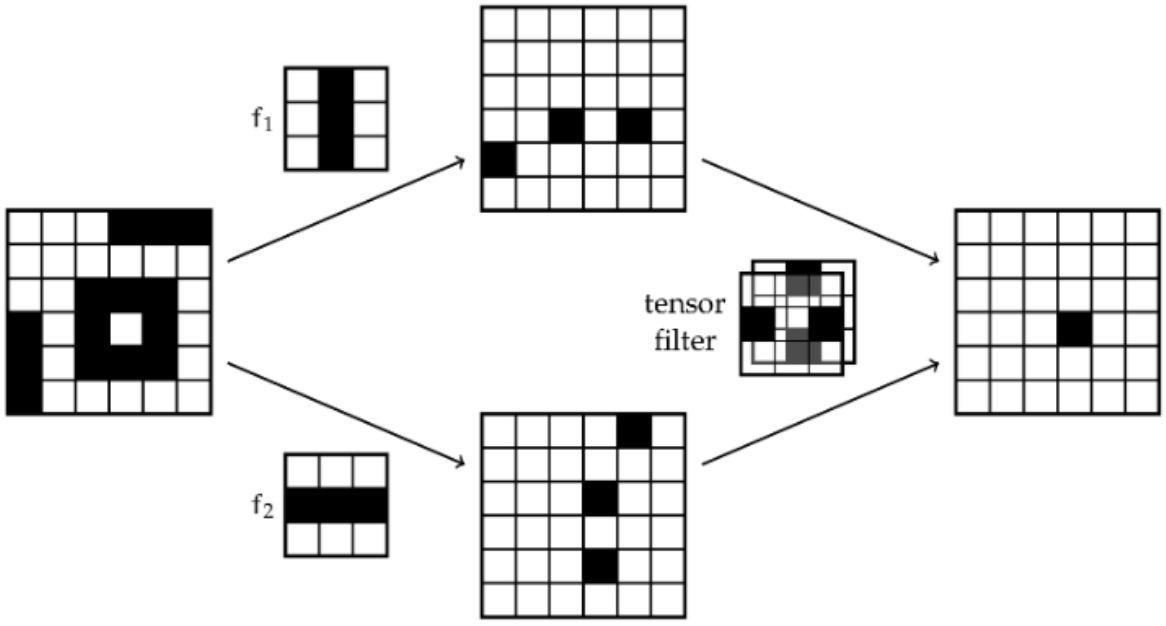


Figure 9: Illustration of two 3×3 filters (vertical and horizontal) producing two feature maps that are stacked into a tensor, followed by a 3D filter extracting combined patterns.

Formal Definition of a Convolutional Layer

A convolutional layer l can be specified by the following hyperparameters:

- **Number of filters:** m^l.
- **Filter size:** $k^l \times k^l \times m^{l-1}$, where m^{l-1} is the number of input channels.
- **Stride:** s^l, the spacing at which the filter is applied.
- **Input size:** $n^{l-1} \times n^{l-1} \times m^{l-1}$.

The output has shape:

$$n^l \times n^l \times m^l, \quad \text{where } n^l = \left\lfloor \frac{n^{l-1}}{s^l} \right\rfloor.$$

Each filter contains $k^l \times k^l \times m^{l-1}$ weights, and there are m^l filters in total.

Key Insight

Thanks to **weight sharing**, CNNs require far fewer parameters than fully connected networks. The same filter weights are reused across the entire image, capturing translation-invariant features.

—

Study Questions

1. **How many weights are there in a convolutional layer with filter size $k^l \times k^l \times m^{l-1}$ and m^l filters?**

   Each filter has $k^l \times k^l \times m^{l-1}$ weights. Since there are m^l filters, the total number of weights is:

   $$m^l \cdot (k^l \times k^l \times m^{l-1}).$$

2. **If we used a fully connected layer with the same input and output dimensions ($n^{l-1} \times n^{l-1} \times m^{l-1}$ inputs, $n^l \times n^l \times m^l$ outputs), how many weights would it have?**

   A fully connected layer would have one weight per input–output connection:

   $$\left(n^{l-1} \times n^{l-1} \times m^{l-1}\right) \cdot \left(n^l \times n^l \times m^l\right).$$

3. **Compare the two cases. Why is the convolutional layer so much more parameter-efficient?**

   In the convolutional layer, the same small set of weights is *shared* across the entire image. In the fully connected layer, each output unit has its own separate set of weights for every input. Thus, convolution drastically reduces the number of parameters, making training faster, less prone to overfitting, and better at capturing local patterns.

Worked Example: Parameter Counts

Let's compare the two cases for a concrete example:

- Input: $32 \times 32 \times 3$ (a small color image).
- Convolutional layer: filter size 5×5, $m^l = 6$ filters.

Convolutional layer weights:

$$6 \cdot (5 \times 5 \times 3) = 6 \cdot 75 = 450 \text{ weights.}$$

Fully connected layer weights:

$$(32 \times 32 \times 3) \cdot (32 \times 32 \times 6) = 3072 \cdot 6144 = 18,874,368 \text{ weights.}$$

Comparison: 450 vs. over 18 million! This dramatic difference illustrates why CNNs are vastly more efficient than fully connected networks for image data.

Hands-on Lab: Filters and Channels in PyTorch

Step 1: Load a real image (CIFAR-10 sample).

```
import torch
import torchvision
import torchvision.transforms as T
import matplotlib.pyplot as plt

# Load CIFAR-10 dataset
transform = T.Compose([T.ToTensor()])
dataset = torchvision.datasets.CIFAR10(root="./data", train=False
    ,
                                       download=True, transform=
                                          transform)
img, label = dataset[0]   # one sample (shape: 3x32x32)

# Show the image
plt.imshow(img.permute(1, 2, 0))  # convert CHW -> HWC
plt.title(f"Label: {label}")
plt.show()
```

Step 2: Apply a convolutional layer.

```
import torch.nn as nn

conv = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5)

# Add batch dimension (1,3,32,32)
x = img.unsqueeze(0)
y = conv(x)

print("Input shape :", x.shape)
print("Output shape:", y.shape)
```

Step 3: Count parameters.

```
def count_params(model):
    return sum(p.numel() for p in model.parameters() if p.
       requires_grad)

print("Convolutional parameters:", count_params(conv))
```

Step 4: Inspect weights.

```
print("Conv weight shape:", conv.weight.shape)
print("Conv bias shape  :", conv.bias.shape)
```

Expected Output (abridged):

```
Input shape :  torch.Size([1, 3, 32, 32])
Output shape: torch.Size([1, 6, 28, 28])
Convolutional parameters: 456
Conv weight shape: torch.Size([6, 3, 5, 5])
```

6 Max Pooling Layers

Definition

A **max pooling layer** is a downsampling operation that reduces the spatial size of feature maps. It has no trainable weights: instead, it slides a window of size $k \times k$ across the input and outputs the maximum value in each region.

Intuition

- Pooling reduces the resolution of the image, building a **feature pyramid**.
- Early layers detect local edges; later layers pool these into larger, more abstract patterns.
- Max pooling achieves a form of **translation invariance**: the exact location of a feature is less important than its presence.

Note

Max pooling is like "summarizing" an image region by its strongest activation. This helps the network generalize and prevents it from overfitting to tiny pixel-level details.

Example

Consider a $64 \times 64 \times 3$ image. A max pooling layer with stride $= k = 2$ will produce a $32 \times 32 \times 3$ output.

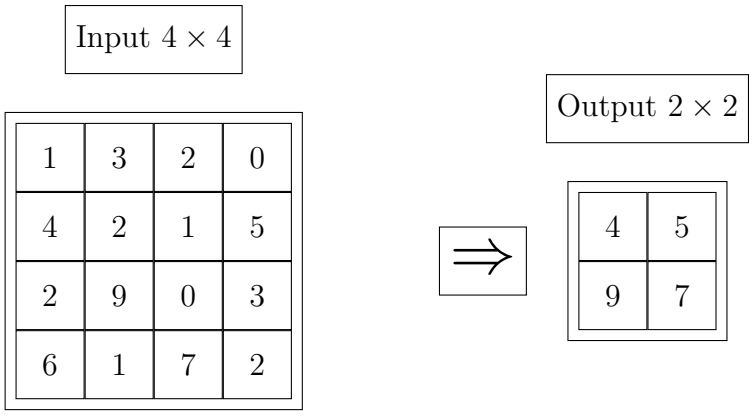


Figure 10: Illustration of max pooling with $k = 2$, stride=2. Each 2×2 block in the input is replaced by its maximum value in the output.

Study Question

Maximilian Poole suggests adding two max pooling layers with size $k = 2$, stride=2, one after the other. What single operation would this be equivalent to?

Example

Answer: It would be equivalent to a single pooling layer with stride = 4. Applying 2×2 pooling twice reduces a 64×64 image first to 32×32, then to 16×16, which is the same as pooling with $k = 4$, stride=4 in one step.

Hands-on Lab: Max Pooling in PyTorch

Step 1: Load an image from CIFAR-10.

```
import torch
import torchvision
import torchvision.transforms as T
import matplotlib.pyplot as plt

# Load CIFAR-10 (test set)
transform = T.Compose([T.ToTensor()])
dataset = torchvision.datasets.CIFAR10(root="./data", train=False
    ,
                                       download=True, transform=
                                          transform)

img, label = dataset[0]   # shape: (3, 32, 32)
plt.imshow(img.permute(1, 2, 0))
plt.title(f"Original CIFAR-10 sample (label={label})")
plt.show()
```

Step 2: Apply max pooling with $k = 2$, stride=2.

```
import torch.nn as nn

pool = nn.MaxPool2d(kernel_size=2, stride=2)

x = img.unsqueeze(0)        # add batch dimension -> (1,3,32,32)
y = pool(x)                 # pooled output -> (1,3,16,16)

print("Input shape :", x.shape)
print("Output shape:", y.shape)

# Visualize pooled image
plt.imshow(y.squeeze().permute(1, 2, 0))
plt.title("After Max Pooling (stride=2, k=2)")
plt.show()
```

Expected Output (abridged):

```
Input shape :  torch.Size([1, 3, 32, 32])
Output shape: torch.Size([1, 3, 16, 16])
```

Discussion:

- The 32×32 image shrinks to 16×16, while keeping 3 channels.
- Max pooling reduces spatial resolution but preserves the strongest activations.
- This operation builds a feature pyramid: early local details $\rightarrow$ larger patterns in later layers.

7 Typical CNN Architecture

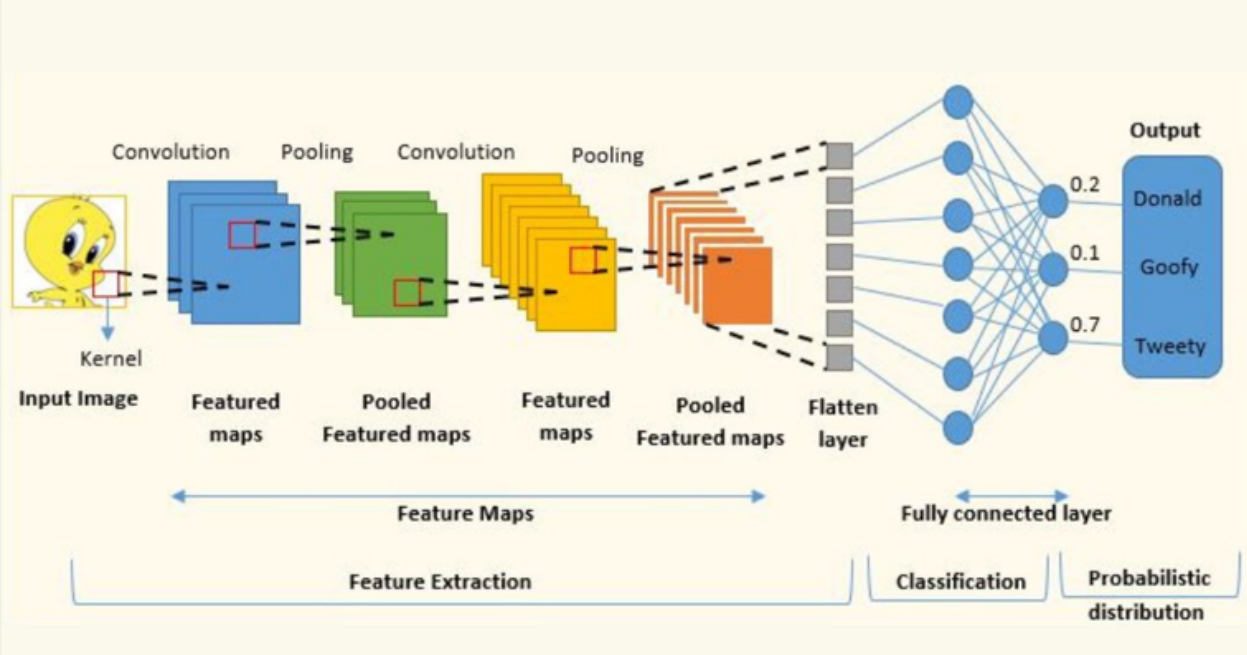


Figure 11: Typical CNN architecture: alternating convolution, activation, and pooling layers, followed by fully connected and output layers. (Adapted from

Definition

A **Convolutional Neural Network (CNN)** typically consists of repeated stacks of convolutional layers (filters), non-linear activations (ReLU), and pooling layers, followed by fully connected layers and an output layer (often softmax for classification).

Layer-by-Layer Structure

1. **Convolution + ReLU:** Detect local patterns (edges, textures). ReLU introduces non-linearity.
2. **Pooling:** Downsample feature maps, summarizing activations and reducing spatial size.
3. **Stacking:** Multiple rounds of Conv+ReLU+Pooling build a hierarchy of features: from edges → shapes → objects.
4. **Fully Connected Layers:** Interpret the learned features as global patterns.
5. **Softmax Output:** Produces class probabilities.

Key Insight

Although CNNs are built from simple blocks (Conv, ReLU, Pooling, FC, Softmax), their composition creates powerful hierarchical representations. Designing the "best" architecture remains part art, part science.

8 Backpropagation in a Simple CNN

Setup

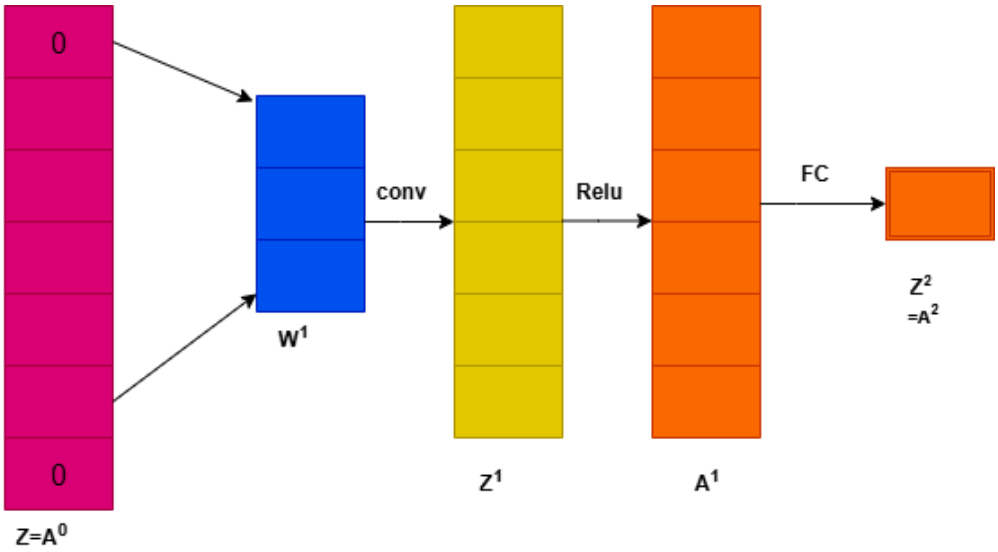


Figure 12: Illustration of backpropagation for a 1D convolutional layer with ReLU activation. Shows dependence of output on filter weights and gradient flow.

We work with a one-dimensional single-channel image, of size $n \times 1 \times 1$. The architecture is:

1. Input image: $X = A^0$.
2. Convolutional layer: a single $k \times 1 \times 1$ filter W^1.
3. ReLU activation.
4. Fully-connected layer with weights W^2, no extra activation.
5. Loss: squared error $L(A^2, y) = (A^2 - y)^2$.

Forward Pass

$$\begin{aligned} Z_i^1 &= (W^1)^T \cdot A^0[i - \lfloor k/2 \rfloor : i + \lfloor k/2 \rfloor] \\ A^1 &= \text{ReLU}(Z^1) \\ A^2 &= (W^2)^T A^1 \\ L(A^2, y) &= (A^2 - y)^2 \end{aligned}$$

Backward Pass

Using the chain rule:

$$\frac{\partial L}{\partial W^1} = \frac{\partial Z^1}{\partial W^1} \cdot \frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial L}{\partial A^1}.$$

- Step 1: Convolution filter dependence

$$\frac{\partial Z_i^1}{\partial W_j^1} = X_{i-\lfloor k/2 \rfloor + j - 1}.$$

- Step 2: ReLU derivative

$$\frac{\partial A_i^1}{\partial Z_i^1} = \begin{cases} 1 & \text{if } Z_i^1 > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Step 3: Gradient wrt A^1

$$\frac{\partial L}{\partial A^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} = 2(A^2 - y)W^2,$$

an $n \times 1$ vector.

Thus, multiplying these components yields the gradient with respect to W^1, of shape $k \times 1$.

Worked Example

If $i = 10$ and $k = 5$, then the entries of column 10 in $\frac{\partial Z^1}{\partial W^1}$ are:

$$[X_8, X_9, X_{10}, X_{11}, X_{12}]^T.$$

This shows how pixel 10 of the output depends on the 5 filter weights.

Study Questions

1. For a filter of size k, how much padding do we need to add to the top and bottom of the image?

2. How is the gradient with respect to W^1 assembled from the local dependencies and the ReLU mask?
3. If we used sigmoid instead of ReLU, how would $\frac{\partial A^1}{\partial Z^1}$ change?

Differentiability and Training

The entire architecture can be seen as a single large differentiable function:

$$f(\mathbf{x}; \mathbf{W}) \to \mathbf{y}$$

where $\mathbf{W}$ are the weights (filters, FC layers). Because the mapping is differentiable, we can compute gradients using backpropagation and optimize the weights using gradient descent.

Study Question

Why do we need non-linear activations (like ReLU) between convolution layers? What would happen if we stacked only convolutional and pooling layers without ReLU?

9 Training CNNs

- Backpropagation through convolution layers (weight sharing).
- Gradient flow through ReLU and pooling.
- Loss: cross-entropy for classification.
- Optimizers: SGD, Adam.

Hands-on Lab

Build and train a small CNN in PyTorch or TensorFlow for MNIST classification.

10 Case Studies

10.1 Why Look at Case Studies?

In this section, we present several case studies of convolutional neural networks (CNNs). The goal is to give intuition on how to effectively combine convolutional layers, pooling layers, and fully connected layers. Just as many learn coding by reading other people's code, one of the best ways to understand CNN architectures is by seeing examples of effective networks.

A neural network architecture that works well for one computer vision task often transfers to other tasks. For instance, a network trained to recognize cats and dogs may be adapted for self-driving car applications. Later sections will also include references to seminal research papers, helping you understand the foundations of modern CNNs.

10.2 Outline of Case Studies

We will examine:

- Classic networks: **LeNet-5**, **AlexNet**, **VGG**
- Modern networks: **ResNet** (Residual Networks), **Inception**

After studying these examples, you will gain intuition for building effective CNNs, and many ideas from these networks have cross-disciplinary applications beyond computer vision.

10.3 Classic Networks

10.3.1 LeNet-5

- Input: $32 \times 32 \times 1$ grayscale image (handwritten digits)
- Layer 1: Convolution with 6 filters of size 5×5, stride 1, output $28 \times 28 \times 6$
- Layer 2: Average pooling, 2×2 filter, stride 2, output $14 \times 14 \times 6$
- Layer 3: Convolution with 16 filters of size 5×5, output $10 \times 10 \times 16$
- Layer 4: Average pooling, output $5 \times 5 \times 16$ (flattened to 400)
- Layer 5: Fully connected, $400 \rightarrow 120$ neurons
- Layer 6: Fully connected, $120 \rightarrow 84$ neurons
- Output: 10-way classification (digits 0–9)

Notes: Height and width decrease as depth increases. Conv + pooling layers often repeat before fully connected layers. Original LeNet-5 used sigmoid/tanh activations and average pooling; modern variants use ReLU and max pooling.

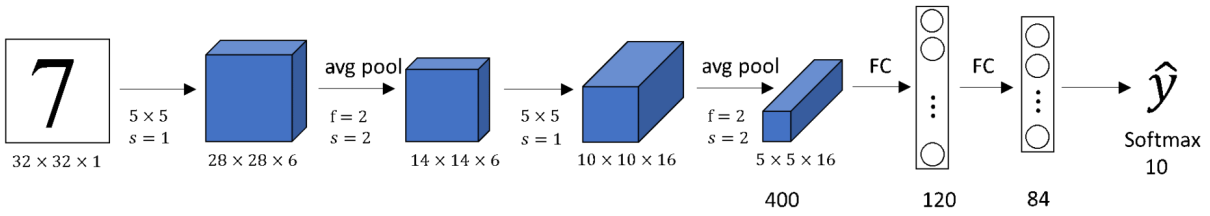


Figure 13: LeNet-5 Architecture. Height/width decrease, channels increase through layers.

11 AlexNet: A Turning Point in Deep Learning

11.1 The ImageNet Competition

In 2012, the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** became a historic moment for deep learning. The task was to classify over **1.2 million high-resolution images** into **1000 categories**, a scale that challenged all traditional computer vision methods based on hand-engineered features such as SIFT and HOG. see the official ILSVRC website: ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

AlexNet, developed by *Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton* at the University of Toronto, achieved a **top-5 error rate of 15.3%**, while the runner-up had 26.2%. This dramatic improvement convinced the computer vision community to take deep learning seriously.

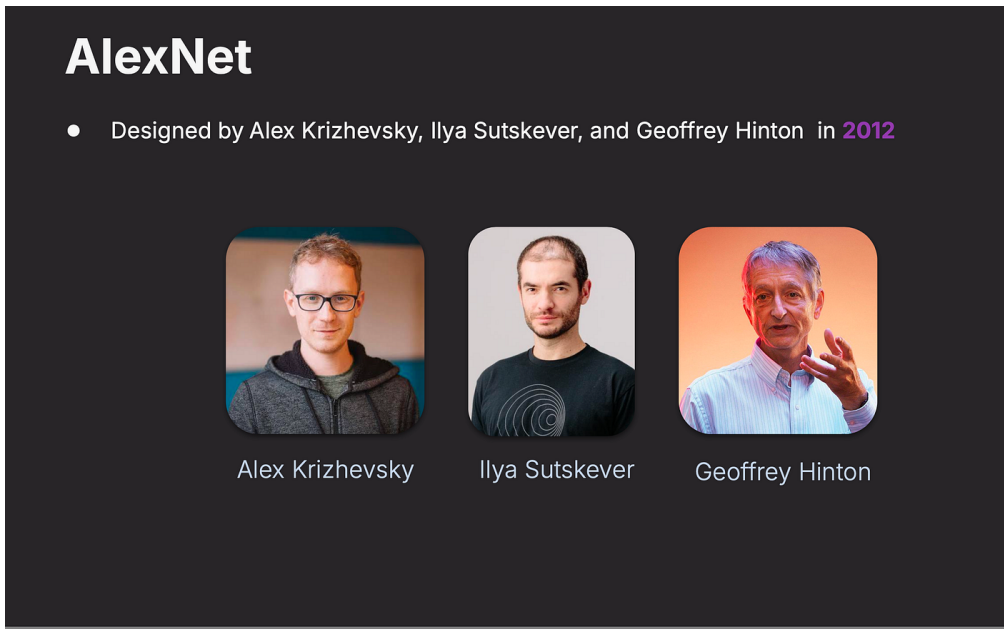


Figure 14: Geoffrey Hinton with Alex Krizhevsky and Ilya Sutskever, the pioneers of AlexNet.

11.2 Why AlexNet Was Revolutionary

Several innovations made AlexNet possible:

- **GPUs for Training**: Accelerated learning on large-scale data.
- **ReLU Activation**: Faster training and mitigation of vanishing gradients.
- **Dropout Regularization**: Reduced overfitting in fully connected layers.
- **Local Response Normalization (LRN)**: Encouraged competition between neurons (later abandoned in modern architectures).

11.3 The AlexNet Architecture

Although AlexNet resembled the earlier LeNet-5, it was significantly larger:

- Input: $227 \times 227 \times 3$ RGB image (paper mentions 224×224, but 227 fits the stride math).
- Conv1: 96 filters of 11×11, stride $4 \rightarrow 55 \times 55 \times 96$.
- Max Pool: 3×3, stride $2 \rightarrow 27 \times 27 \times 96$.
- Conv2: 256 filters of 5×5 (same padding) $\rightarrow 27 \times 27 \times 256$.
- Max Pool: 3×3, stride $2 \rightarrow 13 \times 13 \times 256$.
- Conv3: 384 filters of 3×3 (same padding) $\rightarrow 13 \times 13 \times 384$.
- Conv4: 384 filters of 3×3 (same padding) $\rightarrow 13 \times 13 \times 384$.
- Conv5: 256 filters of 3×3 (same padding) $\rightarrow 13 \times 13 \times 256$.
- Max Pool: 3×3, stride $2 \rightarrow 6 \times 6 \times 256$.
- Flatten: $6 \times 6 \times 256 = 9216$ units.
- Fully Connected Layers: $4096 \rightarrow 4096 \rightarrow 1000$ (softmax).

11.4 Impact

- LeNet-5 (1998) had $\sim 60{,}000$ parameters, while AlexNet had ~ 60 million.
- Its success inspired later architectures such as VGG, GoogLeNet, and ResNet.
- Beyond vision, AlexNet triggered deep learning adoption in NLP, speech recognition, and reinforcement learning.

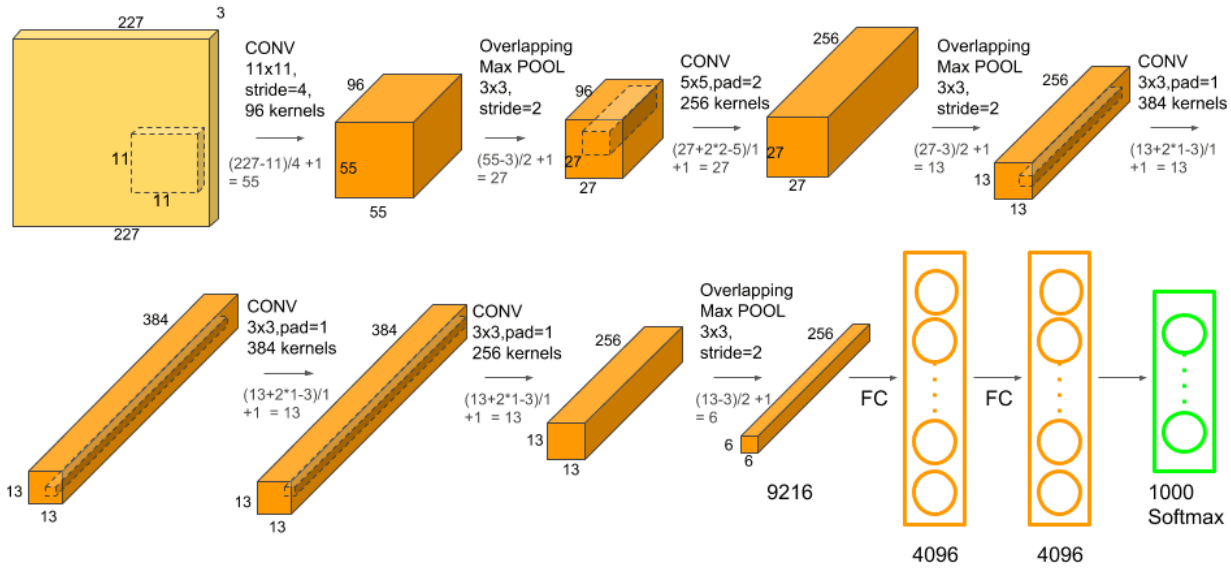


Figure 15: The AlexNet architecture.

11.5 Inspiration

AlexNet shows that breakthroughs often come from scaling existing ideas with larger datasets, more powerful hardware, and innovative training methods.

Exercise: Count AlexNet Parameters

Task. Calculate the number of trainable parameters of AlexNet layer by layer. Recall: - Conv layer: $k_h \times k_w \times C_{in} \times C_{out} + C_{out}$ - FC layer: $N_{in} \times N_{out} + N_{out}$

AlexNet Layers (to use):

1. Conv1: 11×11, $3 \to 96$
2. Conv2: 5×5, $96 \to 256$
3. Conv3: 3×3, $256 \to 384$
4. Conv4: 3×3, $384 \to 384$
5. Conv5: 3×3, $384 \to 256$
6. FC1: $9216 \to 4096$
7. FC2: $4096 \to 4096$
8. FC3: $4096 \to 1000$

Fill in the table:

Layer	Parameters
Conv1	
Conv2	
Conv3	
Conv4	
Conv5	
FC1	
FC2	
FC3	
Total	

Hint: Which layers dominate the parameter count? Why?

Reference

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). *ImageNet Classification with Deep Convolutional Neural Networks*. NIPS. Available at: http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf

12 VGGNet (VGG-16)

Two years after the breakthrough of AlexNet, the Visual Geometry Group (VGG) at Oxford proposed a simpler but much deeper convolutional neural network architecture. In their 2014 paper, *"Very Deep Convolutional Networks for Large-Scale Image Recognition"* (Simonyan & Zisserman), they demonstrated that increasing network depth, while using very small convolutional filters, could significantly improve performance on the ImageNet dataset.

Key ideas:

- All convolutional layers use small 3×3 filters with stride 1 and padding 1.
- Pooling layers use 2×2 max pooling with stride 2.
- Depth: the most popular versions are VGG-16 and VGG-19, with 16 and 19 weight layers respectively.
- A very large number of parameters ($\approx$138 million for VGG-16), making the model computationally expensive.
- Conceptual simplicity: the architecture is just a straightforward stack of convolutions and pooling layers, followed by fully connected layers.

Architecture:

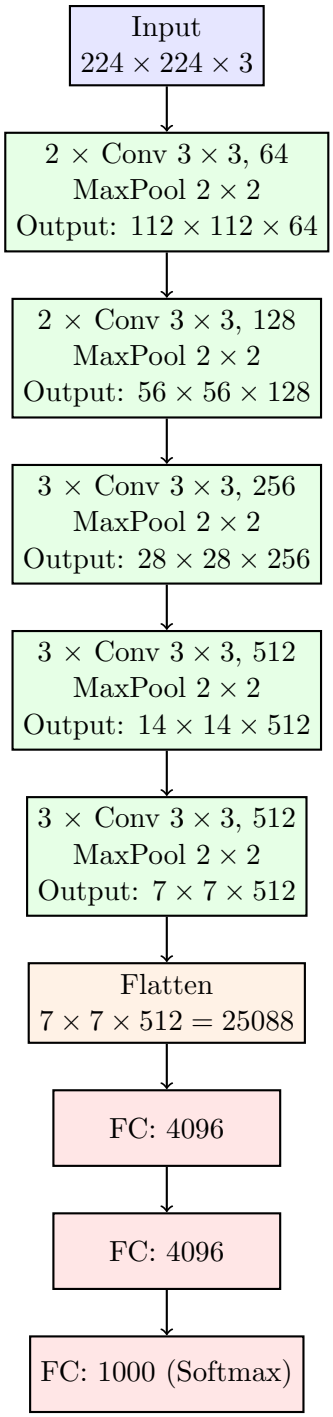


Figure 16: VGG-16 architecture: 13 convolutional layers (3×3 kernels) organized into 5 blocks, followed by 3 fully connected layers.

Why it matters: Despite being heavy in parameters and memory usage, VGG became one of the most influential architectures in computer vision. Its simple design—stacking small filters—established a clear blueprint that influenced many future models. Although

architectures such as GoogLeNet (Inception) performed better in the same ImageNet 2014 competition by being more parameter-efficient, VGG's design had a lasting impact and is still widely used today for transfer learning.

13 Residual Networks (ResNets)

Motivation

Very deep neural networks are notoriously difficult to train due to the problems of **vanishing** and **exploding gradients**. In practice, when the number of layers is increased beyond a certain threshold, training error often worsens instead of improving. In theory, adding more layers should increase the representational power of the network; in practice, however, optimization becomes unstable and performance saturates or degrades.

Residual Block

The key idea of ResNet is the **residual block**. Instead of forcing information to flow strictly through the main sequence of linear and non-linear transformations, ResNets introduce a **skip connection** (or shortcut).

Mathematically, for two layers of computation we normally have:

$$a^{[l+2]} = g(z^{[l+2]})$$

where $z^{[l+2]} = W^{[l+2]}a^{[l+1]} + b^{[l+2]}$.

In a residual block, the activation $a^{[l]}$ is added directly to this computation:

$$a^{[l+2]} = g\big(z^{[l+2]} + a^{[l]}\big)$$

This skip connection allows the block to learn a residual mapping $F(a^{[l]}) := a^{[l+2]} - a^{[l]}$, making it easier for the network to approximate identity functions if needed. Thus, deeper networks do not necessarily degrade in training performance.

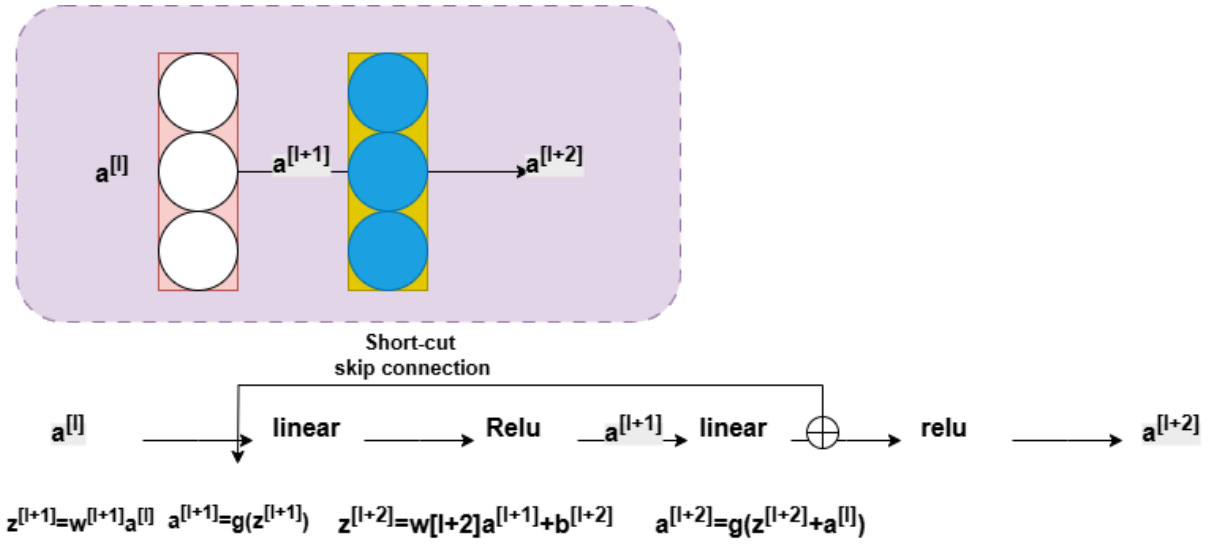


Figure 17: A residual block with a skip connection. Instead of relying solely on the main path, the input activation $a^{[l]}$ is also directly added to the output before the nonlinearity.

Building Residual Networks

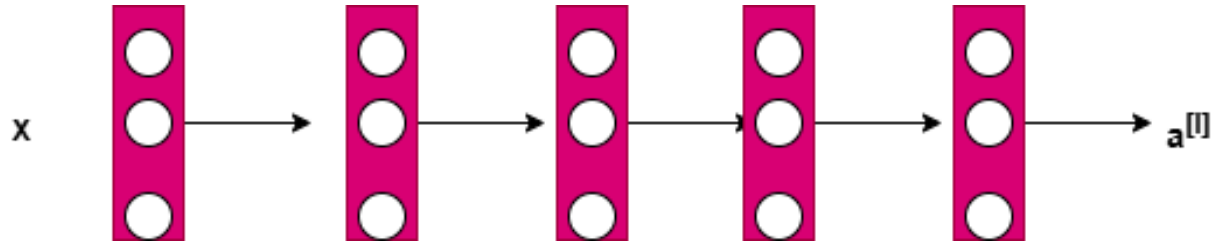


Figure 18: Illustration of the ResNet architecture with stacked residual blocks.

ResNets are constructed by **stacking residual blocks**. For example, the original ResNet-34 is composed of 34 layers of convolution, grouped into stages with shortcut connections linking across two layers at a time.

The figure below contrasts a *plain 34-layer network* with a *34-layer ResNet*. Both have the same number of layers, but the addition of skip connections in ResNet drastically improves trainability.

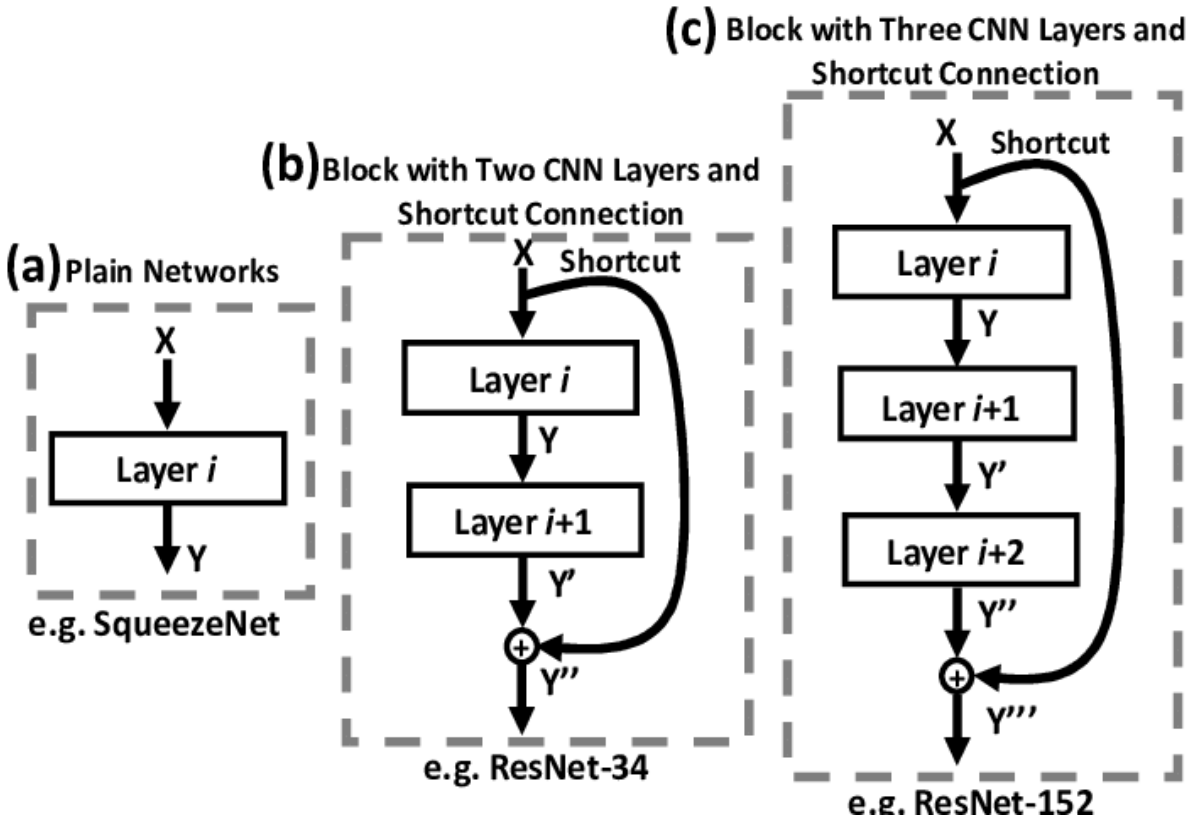


Figure 19: Comparison of a 34-layer plain convolutional network (top) and a 34-layer Residual Network (ResNet) (bottom). In the ResNet, skip connections are added every two layers, enabling much deeper networks to be trained effectively.

Why ResNets Work

The success of ResNets can be attributed to the following:

- **Easier optimization:** residual blocks enable layers to learn small refinements (residuals) rather than the full transformation, making training stable.
- **Identity mapping:** if deeper layers are unnecessary, residual blocks can default to the identity function, ensuring that performance does not degrade.
- **Scalability:** ResNets have been successfully trained with 100, 152, and even 1000+ layers, achieving state-of-the-art results on large-scale vision tasks.

Applications

ResNets form the backbone of many modern computer vision architectures, including detection (Faster R-CNN), segmentation (Mask R-CNN), and beyond. Their success lies in their simple yet powerful design that alleviates vanishing gradients and enables very deep learning.

Active Recall: CNN Architectures

LeNet

Q1: Why is LeNet considered pioneering? **A:** It introduced convolution + pooling + fully connected layers for digit recognition.
Q2: What activation was used in the original LeNet? **A:** Sigmoid/tanh (ReLU came much later).
Q3: What was LeNet's main application? **A:** Handwritten digit recognition on MNIST and bank checks.

AlexNet

Q1: Why did AlexNet win ILSVRC 2012 by a large margin? **A:** Deeper CNN, ReLU activations, dropout, and GPU acceleration.
Q2: What regularization trick did it introduce? **A:** Dropout to reduce overfitting.
Q3: Why was ReLU crucial? **A:** It avoided saturation and sped up convergence.

VGG

Q1: What is the main design principle of VGG? **A:** Use many 3×3 convolutions stacked deep instead of large kernels.
Q2: Why are smaller filters better? **A:** They capture fine features, reduce parameters, and deepen the network.
Q3: What is a limitation of VGG? **A:** High computational cost and memory usage.

GoogLeNet (Inception)

Q1: What problem did Inception modules solve? **A:** How to decide kernel size (1x1, 3x3, 5x5) $\rightarrow$ solution: use all in parallel.
Q2: Why is 1×1 convolution important? **A:** Dimensionality reduction + added nonlinearity.
Q3: How deep was GoogLeNet? **A:** 22 layers with reduced parameters compared to VGG.

ResNet

Q1: Why do residual connections help very deep networks? **A:** They mitigate vanishing gradients and allow identity mapping, enabling training of much deeper networks.
Q2: What happens if a residual block learns zero weights? **A:** The block becomes an identity function, ensuring performance is never worse than a shallower network.
Q3: How are residual blocks typically stacked in ResNet-34? **A:** Skip connections are added every two convolutional layers, grouped into 4 main stages of increasing depth.

14 Practical Tricks & Enhancements

- Data augmentation (flips, rotations).
- Dropout for regularization.
- Batch normalization for stable training.
- Transfer learning with pretrained models (ResNet, MobileNet).

15 Applications

- Image classification (CIFAR-10, ImageNet).
- Object detection (YOLO, Faster R-CNN).
- Semantic segmentation (U-Net).
- Beyond vision: audio spectrograms, text (CNN for NLP).

16 Projects & Assignments

Suggested Projects

1. Train a CNN on CIFAR-10.
2. Fine-tune a pretrained model on a custom dataset (e.g., flowers).
3. Capstone: End-to-end application (medical imaging, face recognition, traffic signs).

17 Practical Tricks & Enhancements

1. Batch Normalization

Definition

Batch Normalization normalizes the output of a layer to have zero mean and unit variance, helping stabilize and accelerate training.

Note

It also reduces sensitivity to initialization and allows for higher learning rates.

Example

Applying batch normalization after each convolutional layer in a CNN often improves accuracy and training speed.

—

2. Dropout

Definition

Dropout randomly sets a fraction of neurons to zero during training, preventing overfitting by forcing the network to learn redundant representations.

Example

A typical dropout rate is 0.5 for fully connected layers and 0.2–0.3 for convolutional layers.

—

3. Data Augmentation Strategies

Definition

Data augmentation generates new training samples by applying transformations to existing data, improving generalization.

Common Techniques

- Random flips and rotations
- Cropping and resizing
- Color jittering (brightness, contrast, saturation)
- Adding noise

Example

Training a CNN on augmented images of cats and dogs reduces overfitting and increases test accuracy.

—

4. Transfer Learning & Pre-trained Models

Definition

Transfer learning uses a model pre-trained on a large dataset (like ImageNet) and fine-tunes it for a new task.

Note

This approach is especially effective when the target dataset is small.

Example

Using a pre-trained ResNet50 and fine-tuning the last few layers for a medical image classification task.

—

5. Learning Rate Scheduling & Optimizers

Definition

Adjusting the learning rate during training or using advanced optimizers can improve convergence.

Popular Strategies

- Optimizers: Adam, RMSProp, SGD with momentum
- Learning rate schedules: Step decay, Cosine annealing, Reduce on plateau

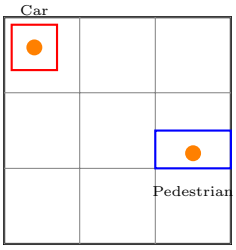
18 Object Detection and YOLO Algorithm

Object detection is the task of identifying and localizing objects in an image. One popular algorithm for real-time object detection is YOLO (You Only Look Once). It divides the image into grid cells and predicts bounding boxes and class probabilities for each cell.

YOLO Grid and Label Assignment

YOLO divides an image into $S \times S$ grid cells. Each grid cell predicts: - Whether an object exists in the cell - The bounding box coordinates (bx, by, BH, BW) - The class of the object

For example, consider a 3x3 grid:



Each grid cell without an object has a label vector with zeros (or "don't cares"). Grid cells with an object encode:

$$Y = \big[\text{Object exists } (1/0), \quad bx, by, BH, BW, \quad \text{class probabilities}\big]$$

For a 3x3 grid and an 8-dimensional label vector per cell, the target output volume is $3 \times 3 \times 8$.

Bounding Box Encoding

Bounding boxes are represented relative to the grid cell: - (bx, by): midpoint of the object, normalized between 0 and 1 within the cell - BH, BW: height and width relative to the cell

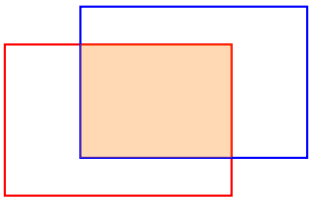
Example: if the midpoint is near the center, $bx \approx 0.5, by \approx 0.5$, and the bounding box may occupy 90% of the cell width: $BW = 0.9$.

Intersection over Union (IoU)

IoU measures overlap between predicted and ground-truth boxes:

$$IoU = \frac{\text{Area of Intersection}}{\text{Area of Union}}$$

- Perfect overlap: $IoU = 1$ - Typically, $IoU \geq 0.5$ is considered correct



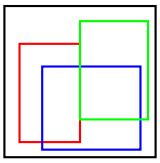
IoU = Intersection / Union

Non-Max Suppression

Multiple predictions may detect the same object. Non-max suppression selects the bounding box with the highest confidence and removes overlapping boxes with IoU above a threshold (e.g., 0.5).

Anchor Boxes

YOLO can detect multiple objects per grid cell using anchor boxes: predefined bounding boxes of different aspect ratios. Each cell predicts adjustments to these anchors to better fit the objects.



Multiple anchor boxes in a cell

Summary

- YOLO predicts objects, bounding boxes, and classes in a single forward pass. - Efficient convolutional implementation allows real-time detection. - IoU, non-max suppression, and anchor boxes improve accuracy and handle multiple objects per cell.

19 Conclusion

Note

This module provides the foundations of CNNs, from intuition and architecture to real-world applications and projects.