Convolutional Neural Networks (CNNs)

Unit 2

Content

Architecture of CNNs (convolution layers, pooling layers, fully connected layers)

Applications in image classification and object detection,

Advanced CNN architectures

(AlexNet, VGGNet, ResNet, InceptionNet)

Transfer learning

Fine-tuning.

Introduction to Convolutional Neural Networks (CNNs

CNNs are designed for grid-structured inputs, primarily images.

They capture spatial dependencies in local regions of the grid.

Images exhibit spatial dependencies where adjacent pixels have similar color values.

CNNs can be applied to text, time-series, and other sequential data.

They are widely used for temporal, spatial, and spatiotemporal data.

CNNs leverage translation invariance, making them suitable for image classification tasks.

Convolution Operation in CNNs

A defining characteristic of CNNs is the **convolution operation**.

Convolution is a dot-product operation between grid-structured weights and input patches.

Useful for data with spatial or locality properties, e.g., images.

CNNs consist of multiple convolutional layers to capture hierarchical features.

Historical Perspective and Biological Inspiration

Evolution of CNN Architectures

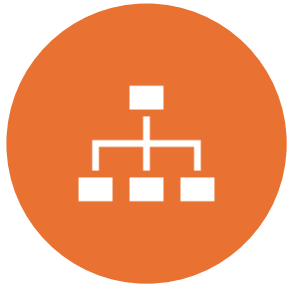
The **Neocognitron** was an early CNN-like model but lacked weight sharing.

LeNet-5 (1998) was the first fully convolutional network, used in digit recognition for bank checks.

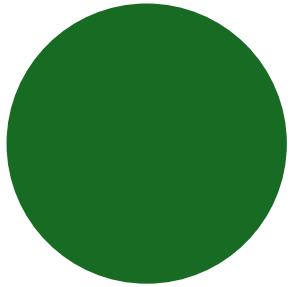
CNNs have evolved mainly in terms of:

- More layers
- Stable activation functions (e.g., ReLU)
- Advanced training techniques and powerful hardware

ImageNet Competition (ILSVRC)



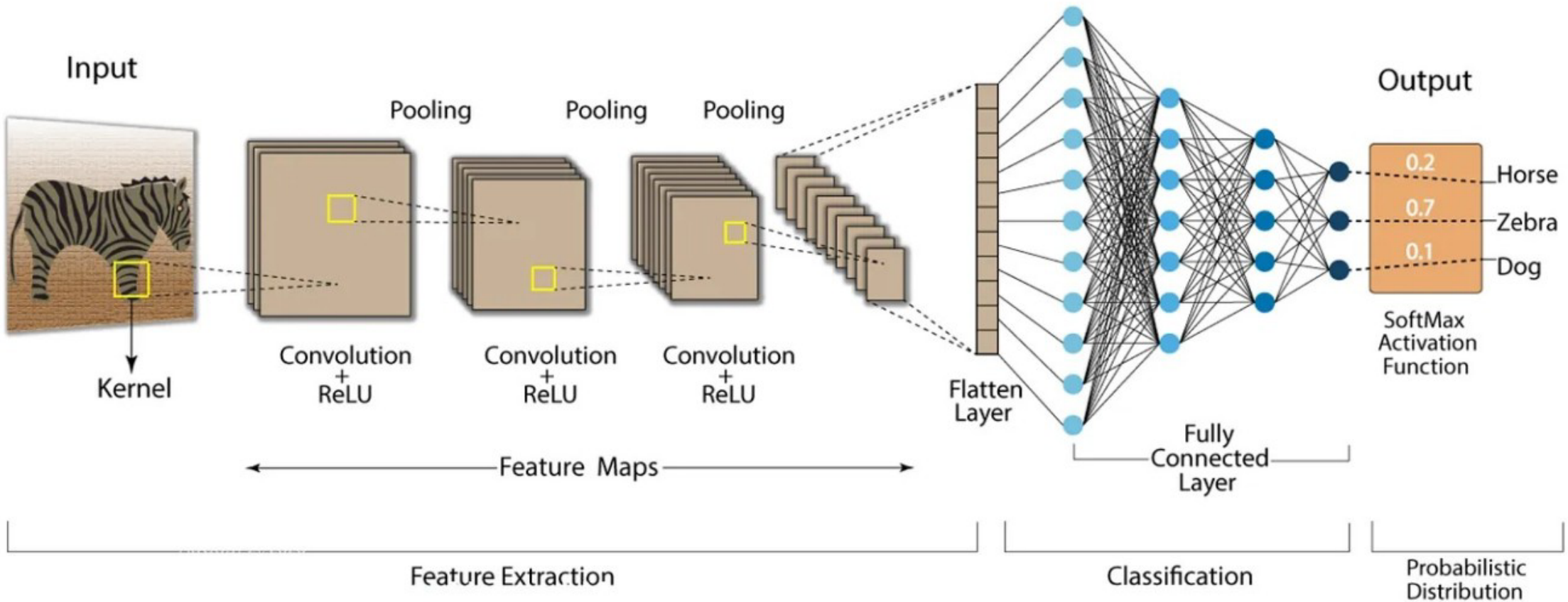
CNNs have dominated image classification since 2012.



AlexNet (2012) was a major breakthrough, winning the ImageNet competition with a large margin.

Improvements between 2012–2015 drastically enhanced accuracy.

Despite small architectural changes, minor details in CNNs significantly impact performance.


Convolution Neural Network (CNN)
Input
Pooling
Pooling
Pooling
Output
0.2
Horse
0.7
Zebra
0.1
Dog
Kernel
Convolution + ReLU
Convolution + ReLU
Convolution + ReLU
Flatten Layer
SoftMax Activation Function
Feature Maps
Fully Connected Layer
Feature Extraction
Classification
Probabilistic Distribution

CNN

https://youtu.be/zfiSAzpy9NM?si=wQ8IoTMQZpjTfRV1

Understanding Convolution in Deep Learning

Convolution is a mathematical operation that combines two functions to produce a third function.

In the context of deep learning, convolution is used to detect patterns such as edges, textures, and shapes in an image.

In simple terms, convolution involves sliding a small matrix (called a filter or kernel) over an input matrix (image) and performing element-wise multiplication followed by summation to obtain a new matrix (feature map or activation map).

Mathematical Representation of Convolution

- If we consider an input matrix and a filter , the convolution operation is given by:

$$O(i,j) = \sum_m \sum_n I(i+m, j+n).F(m,n)$$

- where:
  - $O(i,j)$ is the output at position
  - $I(i+m, j+n)$ represents the region of the image covered by the filter
  - $F(m,n)$ represents the filter weights
- The summation accumulates the product of corresponding values

Example of Convolution

- **Input Image (7 × 7 matrix):**

$$\begin{bmatrix} 6 & 3 & 4 & 4 & 5 & 0 & 3 \\ 4 & 7 & 4 & 0 & 4 & 0 & 4 \\ 7 & 0 & 2 & 3 & 4 & 5 & 2 \\ 3 & 7 & 5 & 0 & 3 & 0 & 7 \\ 5 & 8 & 1 & 2 & 5 & 4 & 2 \\ 8 & 0 & 1 & 0 & 6 & 0 & 0 \\ 6 & 4 & 1 & 3 & 0 & 4 & 5 \end{bmatrix}$$

- **Filter (3 × 3 matrix):**

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

- To compute an output value, place the filter over the top-left corner of the input image, multiply corresponding elements, sum the results, and slide to the next position.
- For one position, the output calculation is:

  5 X 1+ 8 X 1+ 1 X 1 + 1 X 2 = 16
- For another position:

  4 X 1+ 4 X 1+ 4 X 1+ 7 X 2 =26
- This process continues across the entire input image, generating an output feature map.

Properties of Convolution

Translation Equivariance:

If an object in an image shifts, the convolutional output also shifts correspondingly.

Local Receptive Field:

A neuron in a feature map is only connected to a small region of the input image.

Parameter Sharing:

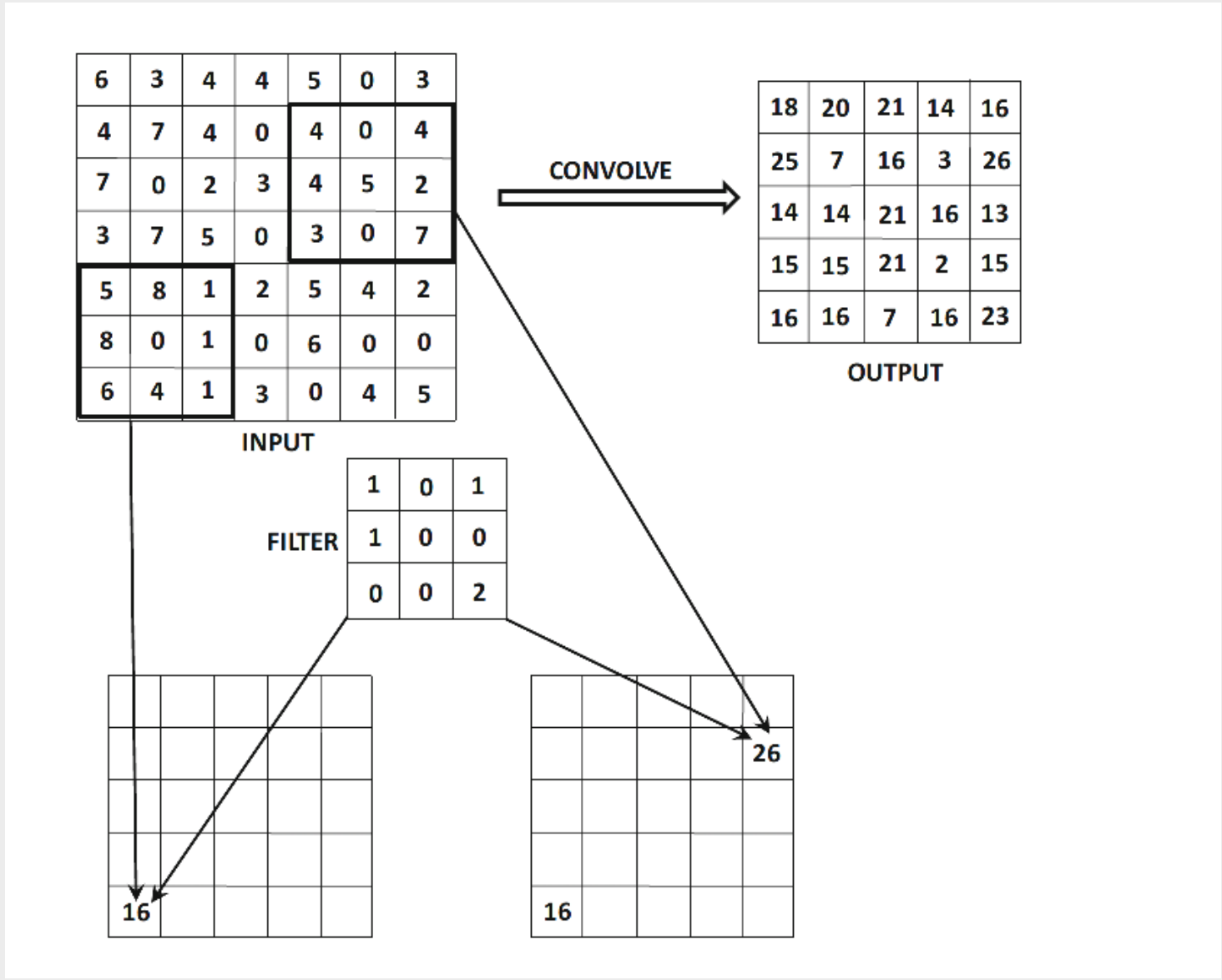
The same filter is applied across the entire image, reducing the number of parameters compared to fully connected networks.

Depth and Multi-Channel Convolution

If an image has multiple channels (e.g., RGB), the filter must have the same depth as the image.

The result of a convolution operation is summed across all channels before passing to the next layer.

Multiple filters produce multiple feature maps, adding depth to the next layer.


6 3 4 4 5 0 3
4 7 4 0 4 0 4
7 0 2 3 4 5 2
3 7 5 0 3 0 7
5 8 1 2 5 4 2
8 0 1 0 6 0 0
6 4 1 3 0 4 5
INPUT
CONVOLVE
18 20 21 14 16
25 7 16 3 26
14 14 21 16 13
15 15 21 2 15
16 16 7 16 23
OUTPUT
FILTER
1 0 1
1 0 0
0 0 2
16
26
16

Implementation in Python (Using NumPy and TensorFlow)

```
import numpy as np
def convolve2D(image, kernel):
    kernel_height, kernel_width = kernel.shape
    image_height, image_width = image.shape
    output_height = image_height - kernel_height + 1
    output_width = image_width - kernel_width + 1
    output = np.zeros((output_height, output_width))

    for i in range(output_height):
        for j in range(output_width):
            output[i, j] = np.sum(image[i:i+kernel_height, j:
j+kernel_width] * kernel)

    return output
```

```
image = np.array([[6, 3, 4, 4, 5, 0, 3],
                  [4, 7, 4, 0, 4, 0, 4],
                  [7, 0, 2, 3, 4, 5, 2],
                  [3, 7, 5, 0, 3, 0, 7],
                  [5, 8, 1, 2, 5, 4, 2],
                  [8, 0, 1, 0, 6, 0, 0],
                  [6, 4, 1, 3, 0, 4, 5]])

kernel = np.array([[1, 0, 1],
                   [1, 0, 0],
                   [0, 0, 2]])

output = convolve2D(image, kernel)
print(output)
```

Output

NumPy Implementation Output

```
[[16 26 18 14 17]
 [22 18 16 19 14]
 [20 15 19 21 18]
 [23 14 22 15 21]
 [15 20 18 24 16]]
```

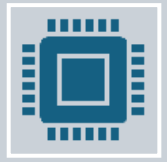
TensorFlow Implementation Output

```
[[[[3.487]     [[3.178]
   [2.971]      [4.125]
   [4.125]      [3.982]
   [3.789]      [4.510]
   [3.210]]     [4.012]]

  [[2.875]
   [3.687]     [[2.876]
   [4.212]      [3.214]
   [3.456]      [3.892]
   [2.983]]     [4.123]
                [3.456]]]]
  [[3.014]
   [3.989]
   [3.765]
   [4.312]
   [3.657]]
```

The shape of the output feature map is (1, 5, 5, 1), which corresponds to a single-channel 5×5 matrix.

Understanding Filters and Depth in Convolutional Neural Networks (CNNs)

Introduction to Convolutional Layers Convolutional Neural Networks (CNNs) are widely used in image processing and computer vision tasks.



The convolutional layer is the core building block of a CNN, responsible for detecting patterns such as edges, textures, and shapes.

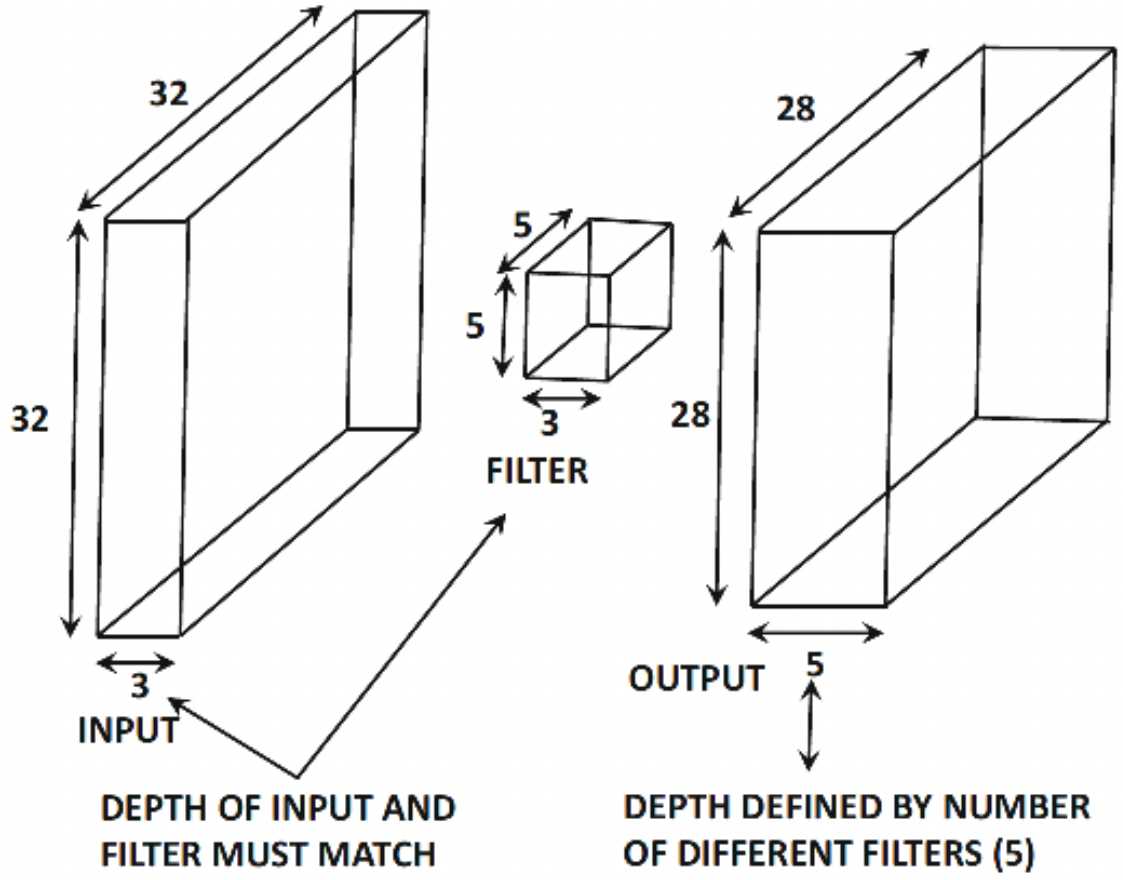
Filters and Depth in CN

A filter (or kernel) is a small matrix that slides over an input image to detect specific patterns.

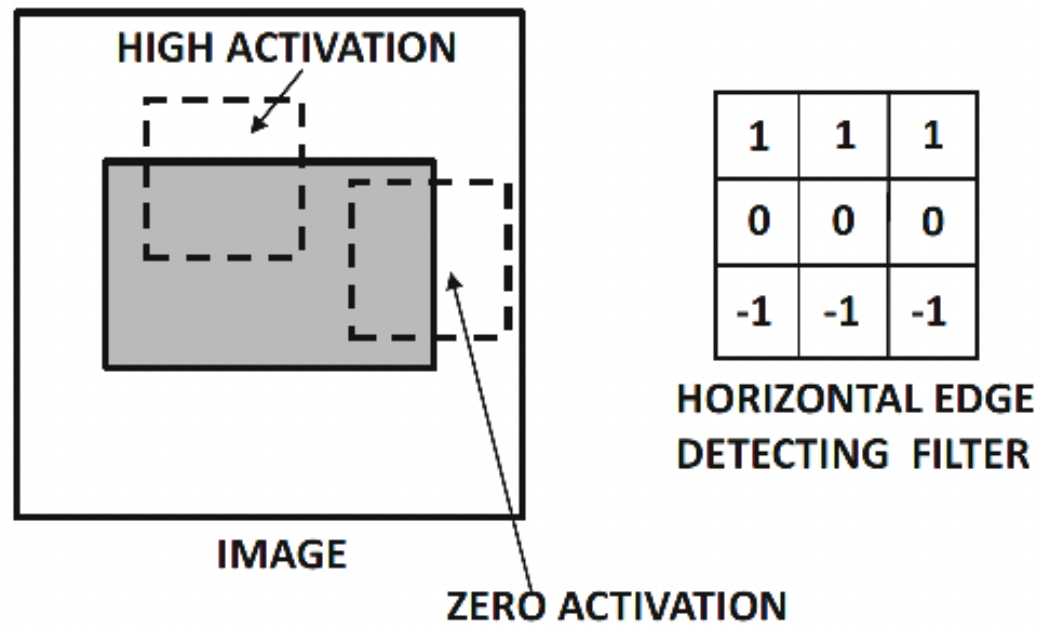
The depth of the input and filter must match. This means that if an input image has three channels (RGB), the filter must also have three channels.

The number of different filters applied to an input determines the depth of the output feature map.

For example, if five filters are applied, the output will have a depth of five.


32
32
3
INPUT
5
5
3
FILTER
28
28
OUTPUT
5
DEPTH OF INPUT AND
FILTER MUST MATCH
DEPTH DEFINED BY NUMBER
OF DIFFERENT FILTERS (5)

(a)


HIGH ACTIVATION
IMAGE
ZERO ACTIVATION
1 1 1
0 0 0
-1 -1 -1
HORIZONTAL EDGE
DETECTING FILTER

(b)

Activation Maps and Edge Detection

Filters are designed to extract specific features from an image.

A horizontal edge detection filter will highlight horizontal lines in an image while suppressing other features.

High activation occurs when the filter detects relevant patterns, whereas zero activation means no feature was detected.

The resulting feature maps are passed to deeper layers for further processing.

Padding

Padding in Convolutional Neural Networks (CNNs) is used to control the spatial dimensions of the output feature maps while preserving essential edge information.

Purpose of Padding:

Prevents the reduction of spatial dimensions after convolution.	Preserves edge features, which might otherwise be lost.	Allows deeper networks to process input data effectively without shrinking.

Types of Padding

Padding Type	Effect
Same Padding	Keeps output size the same, retains spatial information.
Valid Padding	Reduces output size, but avoids artificial boundary artifacts.
Full Padding	Increases output size, useful for edge feature enhancement.

Same Padding (Half-Padding)

Also called **zero-padding**, it ensures the output feature map has the same dimensions as the input.

The number of zeros added on each side is calculated as: P=

where F_q is the filter size.

Example: If the **input size** is 5×5, filter size F_q=3, and stride = 1, then:

- Add **1 row/column of zeros** around the input.
- The output size remains **5 × 5**.

📌 **Representation** (Padding with 1 row/column of zeros):

Input (5x5)		After Padding (7x7)
1 2 3 4 5		0 0 0 0 0 0 0
6 7 8 9 10	→	0 1 2 3 4 5 0
11 12 13 14 15		0 6 7 8 9 10 0
16 17 18 19 20		0 11 12 13 14 15 0

Valid Padding (No Padding)

- No extra zeros are added.
- The feature map size **decreases** after convolution.
- **Example:** For a **5 × 5 input** and a **3 × 3 filter**, with **stride = 1**:
  - Output size = (5−3+1)×(5−3+1)=3×3
  - Output shrinks to **3 × 3**.

Input (5x5)		Output (3x3 after convolution)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	→	8 9 10 13 14 15 18 19 20

Full Padding

- Also called **maximum padding**.
- Adds $F_q - 1$ zeros to **both sides**.
- Increases the spatial footprint.
- **Example:** For a **5 × 5 input** and a **3 × 3 filter**:
  - Add **2 rows/columns of zeros** (because $F_q - 1 = 2$).
- Output **expands**.
  - 📌 **Representation (Padding with 2 zeros around the input):**

Input (5x5)

```
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

→

After Full Padding (9x9)

```
0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0
0  0  1  2  3  4  5  0  0
0  0  6  7  8  9  10 0  0
0  0  11 12 13 14 15 0  0
0  0  16 17 18 19 20 0  0
0  0  21 22 23 24 25 0  0
0  0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0  0
```

Strides in CNNs

Stride refers to the step size by which the convolutional filter moves across the input feature map.

It determines how much the receptive field shifts after each convolution operation.

Effects of Strides

- **Larger strides →**
  - Reduce spatial dimensions, lower computational cost, and provide coarse-grained features.
- **Smaller strides →**
  - Preserve more fine-grained details but increase computational overhead.
- **Strides greater than 1 →**
  - Reduce feature map size, improving memory efficiency and helping prevent overfitting.

Stride Variations : **Stride = 1 (Default)**

- The filter moves **one step at a time**, ensuring maximum feature extraction.
- **Preserves spatial details** but is computationally expensive.
- 📌 **Example:** For a **5 × 5 input**, **3 × 3 filter**, and **stride = 1**, output size:

$$\frac{(5-3)}{1}+1=3\times3$$

🟢 **Advantage:** Captures fine details
🔴 **Disadvantage:** High computational cost

Input (5x5):

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

→

Output (3x3 after convolution)

8	9	10
13	14	15
18	19	20

Stride Variations : **Stride = 2 (Down sampling)**

- The filter moves **two steps at a time**, skipping pixels.
- Reduces spatial dimensions significantly.
- **Often used instead of pooling layers** for down sampling.
- 📌 **Example:**
  - For a **5 × 5 input**, **3 × 3 filter**, and **stride = 2**, output size:

$$\frac{(5-3)}{2}+1=2\times2$$

Input (5x5):					Output (2x2 after convolution)	
1	2	3	4	5 →	8	10
6	7	8	9	10	18	20
11	12	13	14	15		
16	17	18	19	20		
21	22	23	24	25		

🟢 **Advantage:** Reduces dimensions & computational cost

🔴 **Disadvantage:** Some details are lost

Stride > 2 (Extreme Down sampling)

- The filter moves in **larger steps** (e.g., stride = 3 or 4).
- **Rarely used**, only in extreme memory constraints.
- Can lose too much information.
- 📌 **Example:**
  - For a **5 × 5 input**, **3 × 3 filter**, and **stride = 3**, output size:

$$\frac{(5-3)}{3}+1=1\times1$$

Input (5x5):						Output (1x1)
1	2	3	4	5		
6	7	8	9	10	→	8
11	12	13	14	15		
16	17	18	19	20		
21	22	23	24	25		

🟢 **Advantage:** Maximum memory efficiency
🔴 **Disadvantage:** Severe loss of spatial details

Bias in Convolutional Neural Networks (CNNs)

- Bias in CNNs is an additional learnable parameter associated with each filter, added to the result of the convolution operation.
- It helps the network **learn shifts in activation** that might not be captured by weights alone.
- **Mathematical Representation**
  - For an input feature map X, filter weights W, and bias b, the convolution operation is:

$$Y=(X*W)+b$$

where:

- X∗W is the convolution between the input and filter,
- b is the bias term added to each output feature map.

Effect of Bias

Shifts the output activation values

making it easier for the network to learn patterns.

Acts like an additional feature with a constant value of +1,

ensuring that neurons remain active even when all inputs are zero.

Learned during backpropagation,

just like weights, through gradient descent.

Example of Bias :

- **Without Bias (Zero Bias)**
  - If **bias = 0**, then convolution purely depends on weights:

Input (3x3):		**Filter (3x3):**		**Convolved Output:**
1 2 3		1 0 -1		0 0 0
4 5 6	*	1 0 -1	=	0 0 0
7 8 9		1 0 -1		0 0 0

- All zeros mean activation is dead if the sum is 0.
- **Example With Bias = +1**
  - Adding bias helps neurons activate even when inputs sum to zero:

Input (3x3):		**Filter (3x3):**		**Convolved Output (with b = +1):**
1 2 3		1 0 -1		1 1 1
4 5 6	*	1 0 -1	=	1 1 1
7 8 9		1 0 -1		1 1 1

- Activations are shifted, making training more effective.

ReLU Layer in Convolutional Neural Networks (CNNs)

- The **Rectified Linear Unit (ReLU)** is an activation function commonly used in CNNs. It is applied after the convolution operation to introduce non-linearity into the network.
- The function is defined as:

$$f(x) = \max(0,x)$$

  - If $x>0$, it remains the same.
  - If $x\leq 0x$, it is replaced with 0.

Role of ReLU in CNNs

- ReLU is applied element-wise to all values in a feature map.
- It does not change the dimensions of the input.
- It helps CNNs learn complex patterns by introducing **non-linearity**.
- It is faster and more efficient than traditional activation functions like **sigmoid** or **tanh**.
- 📌 **Key Formula:**
  - If an input feature map has size $\mathbf{L_q \times B_q \times d_q}$, after applying ReLU, the output feature map retains the same dimensions:

$$\text{ReLU Output Size} = L_q \times B_q \times d_q$$

Visual Representation of ReLU

- **Input Feature Map Before ReLU:**

```
[[-2,  3, -1],
 [ 4, -5,  6],
 [-7,  8, -9]]
```

- **After Applying ReLU**:

```
[[ 0,  3,  0],
 [ 4,  0,  6],
 [ 0,  8,  0]]
```

- Negative values are set to 0, while positive values remain unchanged.

Why ReLU is Essential in CNNs

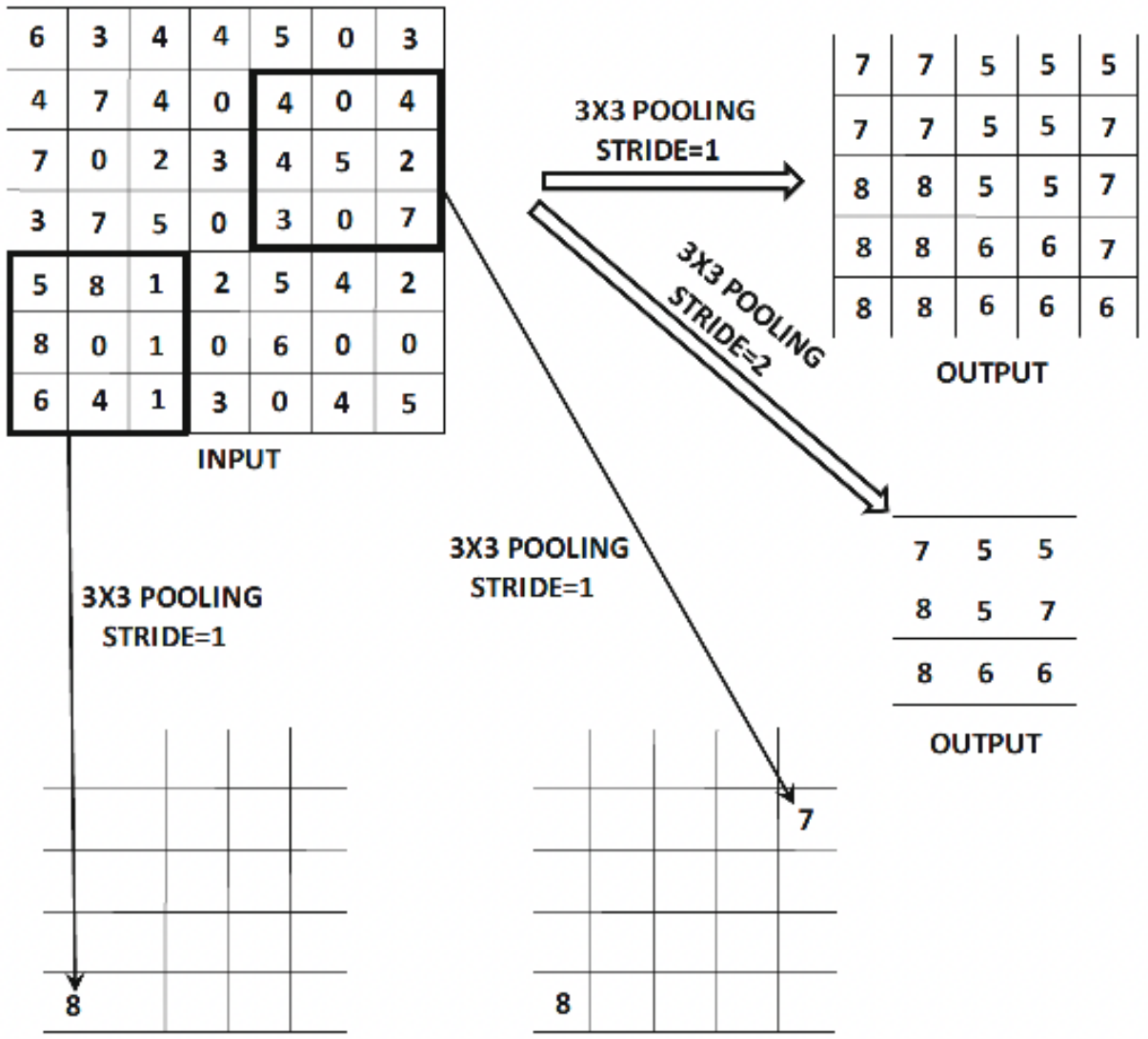
Introduces non-linearity:	CNNs need to capture complex features, and ReLU ensures the network can model non-linear relationships.
Prevents gradient saturation:	Unlike sigmoid/tanh, ReLU does not saturate for large positive values.
Improves training speed:	Avoids expensive exponential computations present in sigmoid/tanh.
Enables deeper networks:	More layers can be stacked without severe performance degradation.

Implementation of ReLU in Python using

```
import numpy as np

def relu(x):
    return np.maximum(0, x)

# Example input feature map
feature_map = np.array([[-2, 3, -1], [4, -5, 6], [-7, 8, -9]])

# Apply ReLU
relu_output = relu(feature_map)
print(relu_output)
```

Max-Pooling in CNNs

- Pooling is a crucial operation in Convolutional Neural Networks (CNNs) that helps **reduce spatial dimensions** while **preserving important features**.
- **Function**:
  - Selects the maximum value from a small region (typically 2×2 or 3×3) of the feature map.
- **Reduces spatial dimensions**:
  - Helps down sample the image while retaining important features.
- **Translation Invariance**:
  - Ensures small shifts in the image do not drastically affect classification.


6 3 4 4 5 0 3
4 7 4 0 4 0 4
7 0 2 3 4 5 2
3 7 5 0 3 0 7
5 8 1 2 5 4 2
8 0 1 0 6 0 0
6 4 1 3 0 4 5
INPUT
3X3 POOLING
STRIDE=1
7 7 5 5 5
7 7 5 5 7
8 8 5 5 7
8 8 6 6 7
8 8 6 6 6
OUTPUT
3X3 POOLING
STRIDE=2
7 5 5
8 5 7
8 6 6
OUTPUT
3X3 POOLING
STRIDE=1
3X3 POOLING
STRIDE=1
8
8
7

Example of a 2×2 Max-Pooling Operation (Stride = 2)

👉 The **largest** value from each 2×2 region is chosen.

nput (4×4)	Max-Pooled Output (2×2)
1 3 2 4	**3 4**
5 6 1 8	**6 8**
9 2 7 3	**9 7**
4 5 6 1	**5 6**

Max-Pooling vs. Strided Convolutions

- 📌 **Conclusion:** While **strided convolutions** can replace pooling, max-pooling introduces **non-linearity** and **greater translation invariance**, making it useful in many architectures.

Feature	Max-Pooling	Strided Convolution
Purpose	Downsampling	Feature extraction + downsampling
Effect	Translation invariance	Feature learning
Non-linearity	Yes	No
Common use	Often used	Used in newer architectures (ResNet, DenseNet)

Step-by-Step Max-Pooling Process

Divide the image into **2×2 non-overlapping regions**.

Pick the maximum value from each region.

Generate a new down sampled feature map.

🔢 Example:

- Original 6×6 Feature Map

```
[[ 1,  3,  2,  4,  8,  6],
 [ 5,  6,  1,  8,  7,  3],
 [ 9,  2,  7,  3,  6,  5],
 [ 4,  5,  6,  1,  9,  2],
 [ 8,  3,  4,  7,  2,  1],
 [ 5,  7,  9,  2,  3,  6]]
```

Max-Pooled Output (3×3)

```
[[ 6,  8,  8],
 [ 9,  7,  9],
 [ 8,  9,  6]]
```

Fully Connected (FC) Layers in Convolutional Neural Networks (CNNs)

- FC layers appear toward the **end** of a CNN and serve as the final decision-making component.
- Their primary role is to process the high-level features extracted by the convolutional and pooling layers and make final predictions.
- **Dense Connectivity:**
  - Each neuron in an FC layer is connected to **every neuron** in the previous layer, making it similar to a traditional **feed-forward neural network**.
- **Parameter-Heavy:**
  - Due to full connectivity, FC layers contain **a large number of parameters**, often contributing the majority of a CNN's total parameters.
  - Example: Two FC layers with **4096 neurons each** would result in over **16 million** parameters.
- **Alternative to FC Layers:**
  - **Global Average Pooling (GAP)** can replace FC layers, significantly **reducing parameter count** and **improving generalization**.
  - Example: **GoogLeNet** (Inception networks) use GAP instead of FC layers.

Activation Functions Used in FC Layers:

Softmax → Multi-class classification (e.g., digit recognition, image classification)

Sigmoid (Logistic) → Binary classification (e.g., cat vs. dog)

Linear Activation → Regression tasks (e.g., age prediction, price estimation)

Interleaving of Layers in CNNs

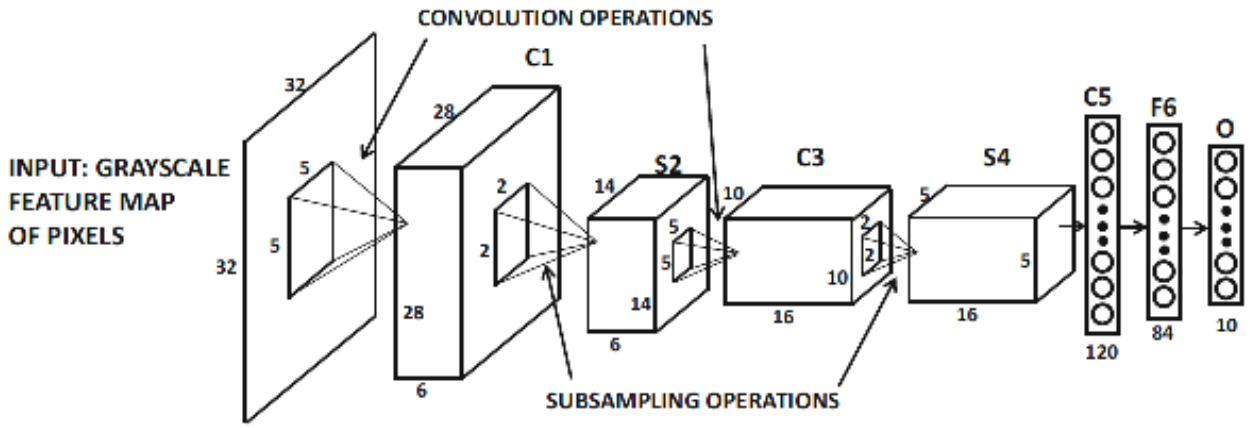
- CNNs are **not just a stack of convolutions**; they follow a structured sequence to progressively extract and refine features.
- **Common Layer Patterns:**
  - A CNN typically follows this pattern:
  - **Convolution (C) → ReLU (R) → Pooling (P)**
- Examples of common interleaving patterns:
  - **CRCRP:** Two convolution layers, each followed by ReLU, then pooling.
  - **CRCRCRP:** Three convolution layers, each followed by ReLU, then pooling.
  - This pattern is often **repeated multiple times** before reaching the FC layers.
- **Pooling Strategies:**
  - **Max-Pooling:**
    - Reduces the spatial size of feature maps.
    - Lowers computational and memory requirements.
    - Extracts the most important features (maximum value in a region).
- **Strided Convolutions:**
  - Sometimes used **instead of pooling** to reduce spatial dimensions while still **learning features**.

LeNet-5 Architecture Explained with the Given Image

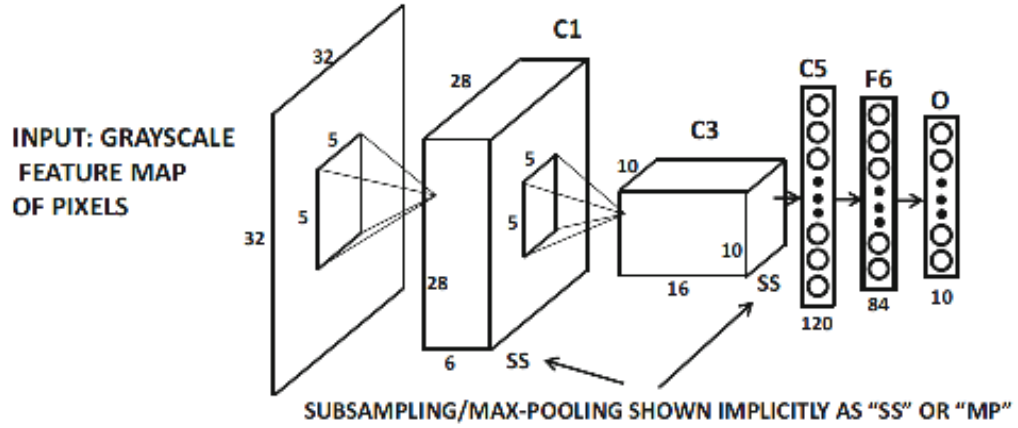
LeNet-5, introduced by **Yann LeCun in the 1990s**, is one of the earliest convolutional neural networks (CNNs).

It was primarily designed for **handwritten digit recognition** (e.g., MNIST dataset).

LeNet-5



(a) Detailed architectural representation



(b) Concise architectural representation

Figure 8.5: LeNet-5: One of the earliest convolutional neural networks.

Architecture

Input Layer

- The network takes a **32×32 grayscale image** as input.
- It processes handwritten digits by extracting hierarchical

Convolutional and Subsampling (Pooling) Layers

- The first few layers extract features and reduce spatial dimensions.
- **C1 (Convolutional Layer)**
  - Uses **6 filters of size 5×5** to generate **28×28 feature maps**.
  - Detects **basic patterns** like edges and textures.
- **S2 (Subsampling / Pooling Layer)**
  - Averages over **2×2 regions** (subsampling operation).
  - Reduces the feature map to **14×14**.
  - Helps in reducing computations and controlling overfitting.
- **C3 (Convolutional Layer)**
  - Uses **16 filters of size 5×5**, producing **10×10 feature maps**.
  - Extracts **more complex patterns** from pooled features.
- **S4 (Subsampling / Pooling Layer)**
  - Again performs **2×2 subsampling**, reducing to **5×5 feature maps**.

Fully Connected Layers for Classification

C5 (Fully Connected Layer, 120 neurons)

Flattens the **5×5×16 feature maps** into **120 neurons**.

Each neuron is connected to all activations from the previous layer.

F6 (Fully Connected Layer, 84 neurons)

Reduces dimensionality before final classification.

O (Output Layer, 10 neurons)

Uses **Radial Basis Function (RBF) units** instead of Softmax.

Outputs a probability distribution over **10 digit classes (0–9)**.

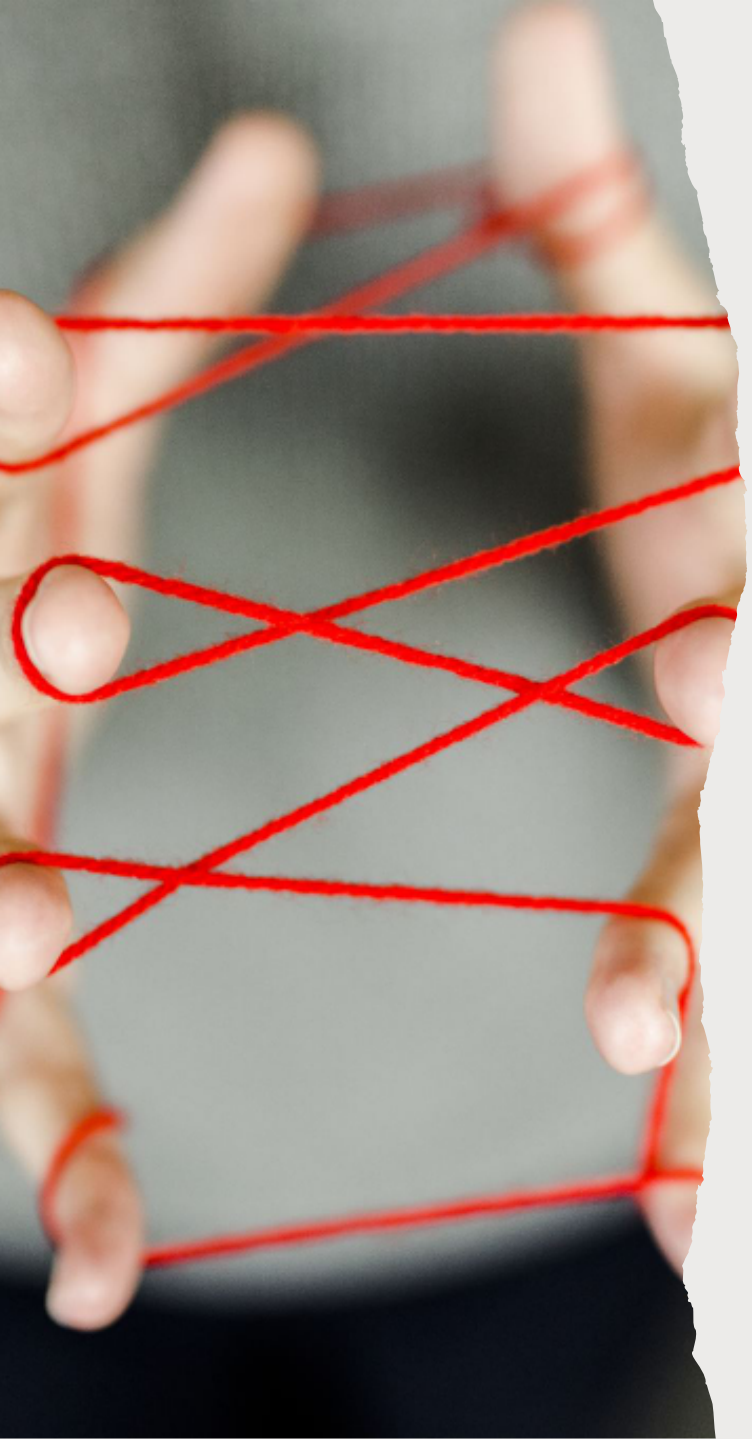
Summary LeNet-5

LeNet-5 was the **foundation** for modern deep learning architectures.

It **extracts hierarchical features** using **convolutional** and **pooling** layers.

It relies on **average pooling** instead of max-pooling (which is common today).

The final classification layer uses **RBF units** instead of Softmax.



Training a Convolutional Neural Network (CNN)

Introduction

Training a CNN involves using the **backpropagation algorithm**, similar to traditional neural networks.

CNNs contain different types of layers: **convolutional layers, activation layers (ReLU), and pooling layers (max-pooling).**

The process of backpropagation varies slightly across these layers.

Backpropagation Through Different Layers

Backpropagation Through ReLU Layer

- The **Rectified Linear Unit (ReLU)** activation function is applied element-wise as:

$$f(x) = \max(0,x)$$

The derivative is straightforward:

- If **x > 0**, the gradient is **1**.
- If **x <= 0**, the gradient is **0** (gradient does not flow backward).

This makes ReLU simple to backpropagate through.

Backpropagation Through Different Layers

Backpropagation Through Max-Pooling Layer

- **Max-pooling** selects the highest value in a local region.
- During backpropagation:
  - The **gradient is assigned only to the maximum value** in the pooling window.
  - All other values in the window receive a **zero gradient**.

Handling Overlapping Pools

- When pooling regions overlap:
  - A neuron may contribute to multiple pooled outputs.
- The gradient contributions from all pools containing that neuron are **summed**.

Backpropagation Through Convolutional Layers

Convolution as Matrix Multiplication

- A convolution operation can be represented as **matrix multiplication**.
- Backpropagation in convolutional layers follows a process similar to fully connected layers but considers **spatial dependencies**.

Element-Wise Backpropagation Approach

- Compute the **loss gradients** for layer .
- Identify the **forward contributions** of each neuron in layer :
  - Each neuron in layer is computed as a weighted sum of a **local receptive field** in layer .
- Each neuron in layer contributes to multiple neurons in layer .
- Compute the **gradients for each neuron in layer ******:
- Backpropagate the error using the **chain rule**.
- The gradient of each neuron in layer is computed as:
- Here, is the **loss derivative** at neuron , and is the corresponding filter weight.

Handling Shared Weights

- Unlike fully connected networks, CNNs use **shared weights** (filters).
- The gradient of a shared weight is computed by:
  - Pretending each filter instance is unique.
  - Summing up the gradients from all positions where the filter was applied.
- This ensures that weight updates are consistent across all spatial locations.

Practical Considerations in CNN Training

- **Weight Initialization**: Proper initialization helps avoid vanishing or exploding gradients.
- **Gradient Clipping**: Used to stabilize training, especially in deep networks.
- **Batch Normalization**: Helps in accelerating convergence.
- **Optimizer Choice**: Adam, RMSprop, and SGD are commonly used.
- **Regularization**: Dropout and L2 regularization prevent overfitting.

Backpropagation as Convolution with an Inverted/ Transposed Filter

Backpropagation is the fundamental mechanism for training neural networks, including Convolutional Neural Networks (CNNs).

When applying backpropagation in CNNs, the weight update process in convolution layers can be interpreted as a convolution operation but with an inverted (or transposed) filter.

This process is crucial in computing the gradients necessary for updating the filter weights.

Understanding Convolutional Forward Propagation

An input image or feature map is convolved with a filter (kernel) to produce an output feature map .

Mathematically, this is expressed as: where denotes convolution.

The convolution operation involves sliding the filter over the input and computing dot products at each step.

Backpropagation in CNNs

- Backpropagation involves computing the gradients of the loss function concerning the parameters of the network.
- For convolution layers, backpropagation consists of two key gradient computations:
  - **Gradient of the Loss with Respect to Input (Error Backpropagation):**
    - The error signal from the next layer is convolved with the transposed filter to distribute the error backward.
  - **Gradient of the Loss with Respect to the Filter Weights:**
    - The gradient of the loss with respect to the filter is computed as a convolution of the input feature map with the error signal.

Convolution as a Transposed Filter in Backpropagation

To understand how convolution appears during backpropagation, we consider two cases:

Gradient w.r.t. Input (Error Propagation Backward)

The backward pass must propagate the gradient to the previous layer's input .

This involves convolving with the spatially flipped version of (i.e., rotated by 180 degrees): where is the transposed (flipped) filter.

Gradient w.r.t. Filter Weights (Weight Update)

The weight gradients are computed as:

Here, is convolved with to obtain the gradient update for .

Transposed Convolution (Deconvolution)

The backpropagation operation in CNNs resembles a **transposed convolution** (or deconvolution), where the dimensions of the output are enlarged compared to the input.

This transposed convolution is used explicitly in applications like image segmentation and generative networks (e.g., GANs, autoencoders).

A transposed convolution essentially redistributes gradient values spatially by spreading them across a larger output space.

Applications

Image Processing:

Understanding transposed convolution is critical in image upsampling techniques.

Neural Style Transfer & Generative Models:

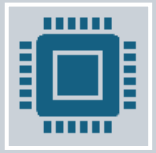
Used in autoencoders and GANs.

Semantic Segmentation:

Used to recover spatial details in segmentation models.

Case Studies from ILSVRC Competition

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has been instrumental in driving CNN advancements.



Several winning architectures have introduced new design principles, making networks more efficient and accurate.

Modern architectures build upon LeNet-5 but with key enhancements to handle large datasets and complex patterns.

Key Advancements in CNN Design

Increase in Depth

- Modern CNNs are significantly deeper compared to early models.
- Example: VGG-16 and ResNet-50 utilize many more layers than LeNet-5, allowing them to capture more complex hierarchical features.
- Deep networks enable better feature extraction but require optimizations to train effectively.

ReLU Activation Function

- ReLU (Rectified Linear Unit) has largely replaced traditional activation functions like Sigmoid and Tanh.
- Benefits of ReLU:
  - Prevents vanishing gradient problem.
  - Accelerates training by introducing non-linearity while keeping computations simple.
  - Enhances gradient flow in deep networks.

Hardware and Computational Improvements

- Modern GPUs and TPUs have significantly boosted training efficiency.
- Compared to early hardware used in LeNet-5, today's platforms are 10,000 times faster.
- Parallel computing and specialized hardware enable efficient training of deep networks with large datasets.

Key Advancements in CNN Design

Training Efficiency and Optimization Techniques

- Use of optimization techniques like Batch Normalization, Adam Optimizer, and Gradient Clipping.
- Dropout and data augmentation techniques prevent overfitting.
- Transfer learning allows models to leverage pre-trained weights, reducing training time and improving accuracy.

Relationship Between Data, Hardware, and Algorithms

- Large-scale datasets are crucial for training deep models.
- Increased computational power enables experimentation with deeper and more complex architectures.
- The combination of high-quality data, efficient algorithms, and modern hardware has fueled the deep learning revolution.

AlexNet

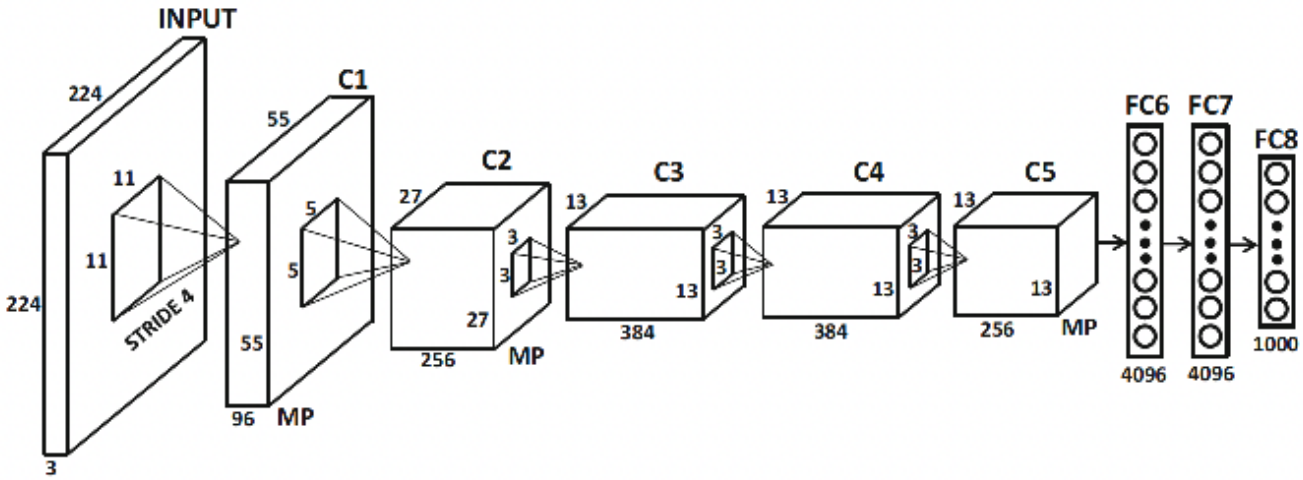
AlexNet was the winner of the 2012 ILSVRC competition

Introduction

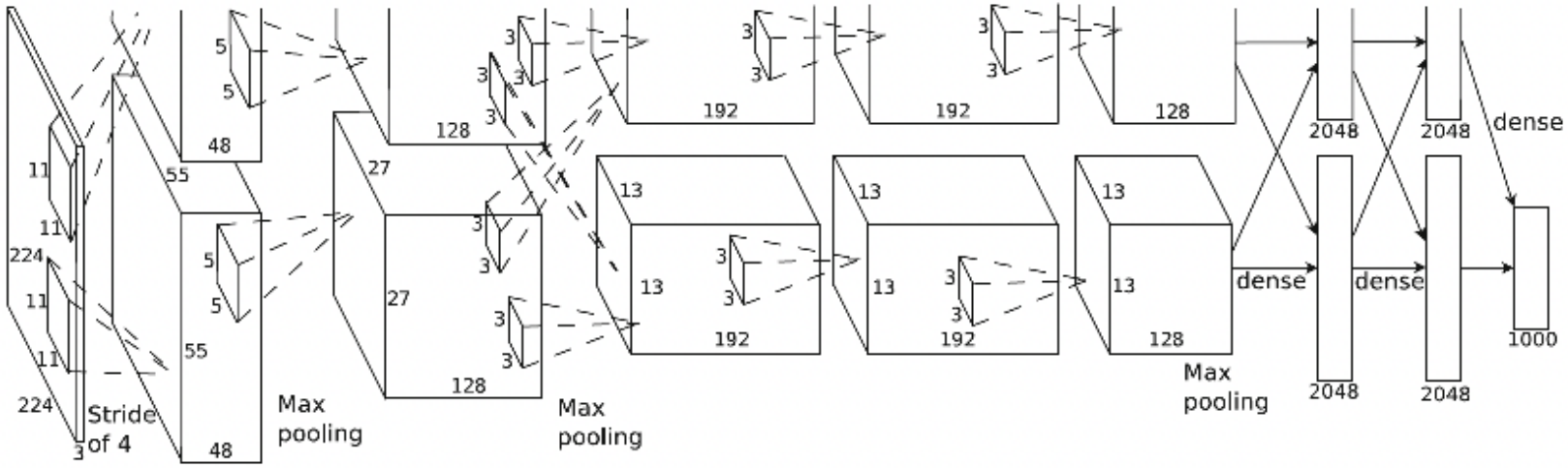
AlexNet is a deep convolutional neural network (CNN) that won the **2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**.

It significantly outperformed previous models, reducing the **top-5 error rate** to **15.4%**, compared to the previous winner's **>25% error rate**.

Key contributions: Demonstrated the effectiveness of deep learning and **popularized CNNs** for image classification.



(a) Without GPU partitioning



(b) With GPU partitioning (original architecture)

AlexNet Architecture

The architecture consists of **8 layers:**

5 convolutional layers (Conv1 to Conv5)

3 fully connected layers (FC6, FC7, FC8, with FC8 using Softmax for classification)

The input is a **224 × 224 × 3** image (RGB).

Uses **ReLU (Rectified Linear Unit)** instead of sigmoid/tanh to speed up training.

Dropout and L2 weight decay were used to improve generalization.

Uses two parallel processing pipelines across two GPUs to handle computational constraints.

Detailed Layer-wise Structure

Layer	Number of Filters	Filter Size	Stride	Feature Map Size
Conv1	96	11 × 11 × 3	4	55 × 55 × 96
Max Pool 1	-	3 × 3	2	27 × 27 × 96
Conv2	256	5 × 5 × 96	1	27 × 27 × 256
Max Pool 2	-	3 × 3	2	13 × 13 × 256
Conv3	384	3 × 3 × 256	1	13 × 13 × 384
Conv4	384	3 × 3 × 384	1	13 × 13 × 384
Conv5	256	3 × 3 × 384	1	13 × 13 × 256
Max Pool 3	-	3 × 3	2	6 × 6 × 256
FC6	4096	-	-	4096
FC7	4096	-	-	4096
FC8 (Softmax output)	1000	-	-	1000

Key Features and Techniques Used in AlexNet

1. GPU-Based Parallel Training

Trained on **two NVIDIA GTX 580 GPUs (3 GB memory each)** due to memory constraints.

Layers were divided between GPUs to **speed up computations**.

2. Activation Function: ReLU (Rectified Linear Unit)

Faster convergence compared to traditional activation functions like sigmoid/tanh.

Prevents vanishing gradient problem.

3. Regularization: Dropout and L2 Weight Decay

Dropout (50% probability): Prevents overfitting by randomly deactivating neurons.

L2 weight decay ($\lambda = 5 \times 10^{-4}$): Reduces the effect of large weight values.

4. Data Augmentation

Used **image translations, horizontal reflections, and RGB intensity variations**.

Helps in increasing dataset size and improving generalization.

5. Local Response Normalization (LRN)

Applied after ReLU in the first two convolutional layers.

Normalizes neuron responses within a small local neighborhood.

(Note: LRN was later abandoned in modern architectures like VGG and ResNet.)

6. Overlapping Max-Pooling

Used **3 × 3 filters** with a **stride of 2** (some overlap).

Helps in **downsampling while retaining spatial features**.

Training Details and Hyperparameters

Optimizer: Stochastic Gradient Descent (SGD) with **momentum 0.8**.

Batch Size: 128

Learning Rate: Initially **0.01**, reduced over time for convergence.

Training Time: **~1 week** (even with GPU acceleration).

Performance and Impact

Top-5 error rate:

- **Single model: 18.2%**
- **Ensemble of 7 models: 15.4%** (winning entry).

Key contributions:

- Popularized CNNs for image classification.
- **Inspired later architectures** like VGG, GoogLeNet, and ResNet.
- **Demonstrated the power of GPUs** for deep learning.
- Introduced widely used techniques like **ReLU, Dropout, and Data Augmentation**.

VGG

VGG Architecture

- VGG [454] contributed significantly to the evolution of deep learning models by emphasizing increased network depth.
- The architecture experimented with network sizes ranging from **11 to 19 layers**, with **16-layer and 19-layer versions performing best**
- **Performance in ISLVRC 2014**
  - **VGG achieved a top-5 error rate of 7.3%**, making it one of the top-performing models.
  - However, **GoogLeNet won** with a lower **top-5 error rate of 6.7%**.
  - Despite not winning, VGG introduced **important design principles** that influenced later architectures.

Key Design Innovations of VGG

Reduction in Filter Size with Increased Depth

- Used **small 3×3 convolutional filters** instead of larger ones.
- Depth was **increased to compensate** for the smaller receptive field.
- Example: Three sequential **3×3 convolutions** capture a **7×7 region** with fewer parameters than a **single 7×7 filter**.

Parameter comparison:

- **Three 3×3 convolutions → 27 parameters (3 × 3 × 3)**
- **One 7×7 convolution → 49 parameters (7 × 7 × 1)**
- Smaller filters also allow capturing **more complex features** through **successive transformations**.

Greater Depth Leads to More Nonlinearity and Regularization

- More **ReLU layers → More nonlinearity** → Better feature extraction.
- Depth forces **structured learning**, leading to **better generalization**.
- **Fewer parameters** required due to smaller filters.

Key Design Innovations of VGG

Use of 3×3 Convolutions and 2×2 Max Pooling

- **Stride = 1, Padding = 1** ensures the **same spatial dimensions** after convolution.
- **Max pooling (stride = 2) reduces spatial dimensions** by half.

Increase in Number of Filters After Max Pooling

- The number of filters **doubled** after each **max-pooling layer**.
- Maintains **computational balance** across layers.
- Later architectures like **ResNet adopted this principle**.

Training Challenges and Solutions

Deeper networks face instability due to sensitivity in weight initialization.

Solution: Pretraining Approach

- Instead of training layer by layer, a **shallow (11-layer) version** was trained first.
- These layers were then used to **initialize deeper networks**.

Spatial Footprint and Computation in VGG

- Max pooling reduces the spatial dimensions as follows:
  - **224 → 112 → 56 → 28 → 14 → 7 (before fully connected layers)**
- At the fully connected layer, the network has a **7×7×512 feature map** connecting to **4096 neurons**.

Parameter and Memory Footprint

Most parameters are in fully connected layers

- Connection from **7×7×512 → 4096 neurons** requires **102.76 million parameters**.
- **Total parameters in VGG: ~138 million**.
- **75% of parameters are in a single fully connected layer**.
- **90% of parameters are in dense connections**.

Memory Requirements

- Early layers require **high memory** due to **large spatial footprints**.
- Example: **93MB per image** → For a **mini-batch of 128**, **12GB memory** is needed.

Filters

Use of 1×1 Convolutions

- Though not spatially combining features, **1×1 convolutions** help:
  - **Merge features from different channels**.
    - **Increase nonlinearity** through additional ReLU activations.
- **Impact of VGG on Future Architectures**
- **Inspired ResNet and other deep models** with its **depth-focused approach**.

Reinforced the effectiveness of small filters (3×3).

- Demonstrated that **depth + nonlinearity** improves feature extraction and classification accuracy.

ResNet

Residual Networks

Residual Networks

ResNet (Residual Networks) introduced in 2015.

Achieved **human-level performance** in image classification with a **top-5 error of 3.6%** in the ILSVRC competition.

Used up to **152 layers**, significantly deeper than previous architectures.

Challenges of Training Deep Networks

Training challenges

- Deep networks suffer from **vanishing/exploding gradients**, making training difficult.
- Batch normalization helps but does not fully solve convergence issues.
- Traditional deep networks force all concepts to be learned at the same depth, slowing convergence.

Skip Connections

- **Skip (Residual) connections** allow the network to bypass layers by directly adding input from layer **i** to layer **(i+r)**.
- Helps in efficient **gradient flow**, reducing the vanishing gradient problem.
- Allows layers to learn **residual functions** instead of the full mapping.

Residual Module (Basic Unit of ResNet) and architecture

Each block contains a **skip connection** that enables easy propagation of gradients.

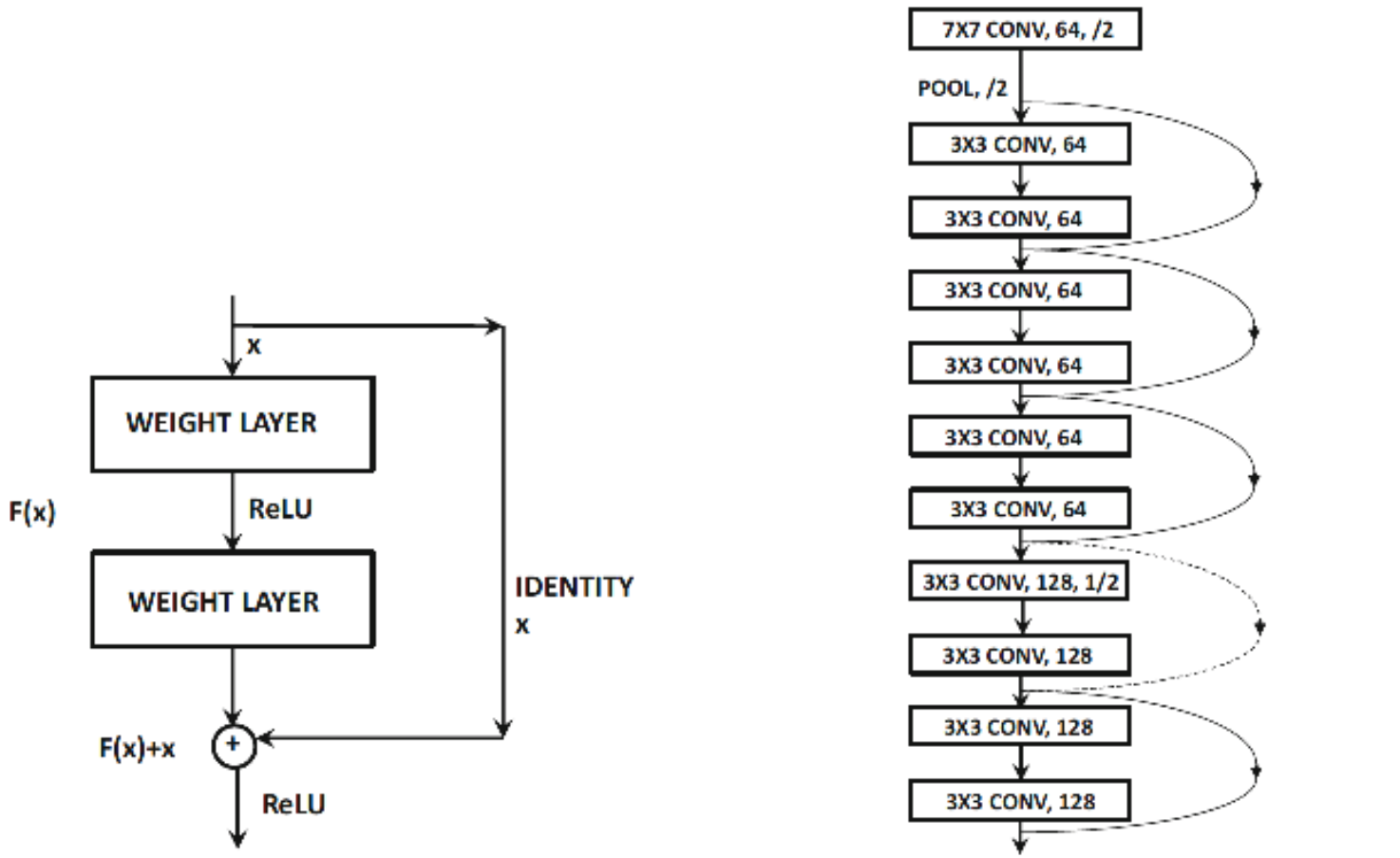
Standard layers use a **stride of 1** to maintain input size.

If a layer reduces spatial dimensions (e.g., stride of 2), a **projection matrix** (1×1 convolution) is used to adjust dimensions.

Four main versions: **34, 50, 101, and 152 layers**.

Even the **34-layer model outperformed previous state-of-the-art models**.

Skip connections provide **multiple learning paths**, acting like an ensemble of shallow networks.



(a) Skip-connections in residual module (b) Partial architecture of *ResNet*

Figure 8.11: The residual module and the first few layers of *ResNet*

Impact on Backpropagation & Learning

Skip connections create **multiple paths of varying lengths** for gradient flow.

Shorter paths contribute more to learning, while longer paths fine-tune the model.

Avoids overfitting by allowing necessary complexity without redundant depth.

Variants and Improvements

DenseNet: Extends ResNet by adding connections between all layers.

Wide ResNet (WRN): Uses fewer layers with wider channels, achieving better performance.

Highway Networks: Uses **gated skip connections**, related to LSTMs.

Image Classification using Convolutional Neural Networks (CNNs)

Image classification is the task of assigning predefined labels (categories) to an image.

This is a core application of **Convolutional Neural Networks (CNNs)**, which are highly effective in **feature extraction and pattern recognition**.

Example:

Input: Image of a cat

Output: "Cat" label assigned to the image

Use Cases:

Identifying objects in images (e.g., cat, dog, car)

Automating visual inspections (e.g., manufacturing defects)

Organizing large image datasets (e.g., Google Photos categorization)

How CNNs Work for Image Classification

- **Input Layer**
  - Receives an image (e.g., 224×224 pixels, RGB).
  - Converts it into numerical pixel values.
- **Convolutional Layers**
  - Extracts features like edges, textures, and patterns.
  - Uses filters (kernels) to scan the image and create **feature maps**.

Filter Type	Purpose
Edge Detection	Identifies object boundaries
Texture Recognition	Detects surface details
Shape Recognition	Recognizes object structure

How CNNs Work for Image Classification

Pooling Layers (Downsampling)

- Reduces image size while preserving important information.
- Improves computational efficiency and reduces overfitting.

Types of Pooling:

- **Max Pooling:** Keeps the highest value from a region
- **Average Pooling:** Takes the average of the region

Fully Connected (FC) Layers

- Converts extracted features into a **final prediction**.
- Uses **Softmax Activation** to assign probabilities to each class.

Image Classification Workflow

Data Collection: Gather labeled images for training.

Example: Cat vs. Dog dataset.

Preprocessing: Resize, normalize, and augment images.

Normalization: Scale pixel values between 0 and 1.

Data Augmentation: Rotate, flip, and zoom to improve model generalization.

Model Selection: Choose a CNN architecture (e.g., ResNet, VGG).

Training: Feed images into the CNN, adjust weights using **backpropagation and optimization** (e.g., Adam, SGD).

Validation & Testing: Evaluate model performance on unseen images.

Use **accuracy, precision, recall, and F1-score** as metrics.

Deployment: Deploy the trained model for real-world applications.

Object Detection & Recognition using CNNs

Object detection and recognition is a key computer vision task that not only classifies objects in an image but also **localizes them** by drawing bounding boxes around them.

Unlike image classification (which assigns a single label to an image), **object detection identifies multiple objects along with their positions**.

Difference Between Object Detection & Classification

- identifies and locates multiple objects in an image.
- Outputs **bounding boxes** with labels and confidence scores.

Task	Output
Image Classification	"This image contains a cat"
Object Detection	"There is a cat at (x1, y1, x2, y2)"
Object Recognition	"This is a Siamese cat at (x1, y1, x2, y2)"

Object Detection Models Explained

Faster R-CNN (Two-Stage Detector)

- **Stage 1: Region Proposal Network (RPN)**

Identifies **potential object regions** in an image.

- **Stage 2: Fast R-CNN for Classification & Refinement**

Classifies objects and fine-tunes bounding boxes.

Pros:

- Highly accurate
- Works well for complex images

Cons:

- Computationally expensive
- Not real-time

YOLO (You Only Look Once) – Single-Stage Detector

Divides an image into grids and predicts objects & bounding boxes in one forward pass.

Uses a **single neural network**, making it extremely **fast**.

Predicts **multiple objects** simultaneously.

- **Pros:**

Real-time performance (high FPS)
End-to-end learning (no need for separate region proposals)

- **Cons:**

Less accurate for small objects
Struggles with overlapping objects

Example: YOLO in Action

Input Image: A street scene with cars and pedestrians.
Output: Bounding boxes around **cars (car, 95%)** and **people (person, 87%**

Object Detection Workflow

Data Collection:	Collect images with labeled bounding boxes (COCO, PASCAL VOC datasets).
Preprocessing:	Resize images, normalize pixel values, and perform data augmentation.
Feature Extraction:	Use a CNN (ResNet, MobileNet) as a **backbone network**.
Bounding Box Prediction:	Model predicts object classes and their positions.
Training & Optimization:	Loss function minimizes classification errors and bounding box errors.
Evaluation:	**mAP (mean Average Precision)** is used as a metric.
Deployment:	Model runs on edge devices, cloud servers, or embedded systems.

Challenges & Future Improvements

- **Challenges in Object Detection**

Occlusion: Objects blocking each other can reduce accuracy.

Real-Time Constraints: Faster models sacrifice accuracy.

Class Imbalance: Some objects appear less frequently in datasets.

- **Future Trends**

Transformers in Object Detection (DETR, Vision Transformers)

Edge AI: Running detection models on mobile devices.

Better Small Object Detection: Improvements in feature pyramids.

Transfer Learning in Deep Learning

🔹 Instead of training a model from scratch, we take a **pretrained model** (e.g., ResNet, VGG) and **fine-tune it** on a smaller dataset.

🔹 Works well when **labeled data is limited** but we have a **large dataset for pretraining**.

🔹 Saves **time, computational resources, and improves accuracy**.

Example:

A model trained on **ImageNet (1.2M images, 1000 classes)** can be fine-tuned for:
✔️ Medical Imaging (X-ray classification)
✔️ Industrial Defect Detection
✔️ Animal Species Recognition

Why Use Transfer Learning?

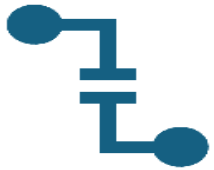
Reduces training time – No need to train from scratch.

Requires less data – Pretrained models generalize well with limited data.

Improves accuracy – Leverages rich feature representations.

Avoids overfitting – Beneficial for small datasets.

Types of Transfer Learning



(a) Feature Extraction

Use a **pretrained model as a feature extractor**.

Remove the last layer and add a new classifier.

Train only the new layers; keep pretrained layers **frozen**.

- **Example:**

- Take **VGG16 trained on ImageNet**.
- Replace the last fully connected layer with a **new classifier**.
- Train only the new classifier on **medical images**.

(b) Fine-Tuning

Instead of keeping pretrained layers frozen, **some layers are trained** on new data.

Helps the model adapt better to domain-specific features.

- **Example:**

- **Fine-tune ResNet** on **self-driving car images** by unfreezing deeper layers.
- The model adapts its features to **road signs and vehicles**.

How Transfer Learning Works?

Step 1: Choose a Pretrained Model

Popular pretrained models include:

ResNet, VGG, Inception, MobileNet (for image tasks)

BERT, GPT, RoBERTa (for NLP tasks)

Step 2: Modify the Model

Remove last fully connected layer (original classifier).

Add new layers to match the target task (e.g., 3-class classification).

Freeze or unfreeze layers based on whether you want to fine-tune.

Step 3: Train on New Dataset

Use **smaller learning rates** for fine-tuning.

Apply **data augmentation** to prevent overfitting.

Applications of Transfer Learning

Image Recognition: Medical imaging, defect detection.

Natural Language Processing: Sentiment analysis, chatbots (BERT, GPT).

Speech Processing: Voice assistants (Google Assistant, Siri).

Autonomous Vehicles: Object detection in self-driving cars.

Finance & Fraud Detection: Credit card fraud detection.

Fine-Tuning in Deep Learning

- In transfer learning, pretrained models are used as **feature extractors**.
- By default, these models have **frozen weights** (i.e., they don't update during training).
- Fine-tuning involves **unfreezing some deeper layers** and training them on the target dataset.
- This allows the model to **adapt to new data distributions**.
- **Example:**
  - A **ResNet50 model pretrained on ImageNet** can be fine-tuned for:
  - **X-ray classification** (medical images).
  - **Self-driving cars** (object detection in traffic scenes).
  - **Satellite image analysis** (land classification).

Why Fine-Tune a Model?

Leverages general knowledge from a large dataset.

Improves performance on domain-specific tasks.

Requires less data than training from scratch.

Speeds up convergence and reduces overfitting

When to Use Fine-Tuning?

Use Fine-Tuning If:

- Your dataset is **small** but related to the pretrained model's domain.
- The pretrained model's features are **useful but need adaptation**.
- More **training time and computational power** are available.

Avoid Fine-Tuning If:

- Your dataset is **too small** (risk of overfitting).
- The pretrained model is **already performing well** as a feature extractor.
- You want **quick results with minimal computation**.

Steps for Fine-Tuning a Pretrained Model

Step 1: Choose a Pretrained Model

Popular pretrained models include:

VGG16, ResNet, Inception, MobileNet (for images)

BERT, GPT, RoBERTa (for NLP)

Step 2: Modify the Model

Remove the last classification layer of the pretrained model.

Add a new classifier (matching the number of classes in your dataset).

Freeze early layers (general features) and **unfreeze deeper layers** (task-specific features).

Step 3: Train on New Dataset

Use **lower learning rates** (to prevent catastrophic forgetting).

Apply **data augmentation** (to improve generalization).

Fine-tune for a **few epochs**, then evaluate performance.

Best Practices for Fine-Tuning

Start with Frozen Layers:

Begin with **feature extraction** (frozen pretrained layers).

Train only the new classifier first.

Unfreeze Few Layers at a Time:

Gradually **unfreeze deeper layers** for fine-tuning.

Prevents large weight updates that can destroy learned features.

Use a Small Learning Rate:

A lower LR (e.g., 0.0001) helps **gradual adaptation**.

Too high an LR may cause **catastrophic forgetting**.

Apply Data Augmentation:

Prevents overfitting on small datasets.

Common techniques: **rotation, flipping, zooming, color jittering**.

Use Early Stopping & Regularization:

Avoid overfitting by using **dropout, L2 regularization, and early stopping**.

Applications of Fine-Tuning

Medical Imaging:	Fine-tune ResNet for **X-ray/MRI classification**.
Autonomous Vehicles:	Adapt YOLO/Faster R-CNN for **pedestrian detection**.
Natural Language Processing (NLP):	Fine-tune BERT/GPT for **text sentiment analysis**.
Industrial Defect Detection:	Train Inception to detect **manufacturing defects**.
Wildlife Conservation:	Fine-tune MobileNet to **classify animal species** from camera trap images.