Fundamentals of Convolutional Neural Networks

Naman Goel

January 7, 2026

Abstract

Convolutional Neural Networks exploit the spatial structure inherent in images through weight sharing and local receptive fields. This report provides a comprehensive treatment of the convolution operation, filter design and learning, pooling mechanisms, receptive field analysis, and backpropagation through convolutional layers.

1 Introduction

Fully connected neural networks treat input dimensions independently without regard to spatial relationships. In image data, this is inefficient: nearby pixels tend to be strongly correlated, and distant pixels are largely independent. Convolutional neural networks exploit this spatial structure through two key mechanisms: (1) local receptive fields that process small regions of the image, and (2) weight sharing where the same filter is applied across all spatial locations.

These mechanisms dramatically reduce parameters compared to fully connected architectures while improving generalization through implicit regularization and translation invariance.

2 The Convolution Operation

2.1 Two-Dimensional Discrete Convolution

The fundamental operation in CNNs is 2D discrete convolution. An input feature map (or image) is processed by a small kernel (filter) that slides across spatial dimensions. At each position, the operation computes a weighted sum of the input region under the kernel.

For an input matrix and a kernel matrix, the convolution at position (i, j) computes:

The kernel is placed at position (i, j), element-wise multiplied with the underlying input region, and the results are summed. This scalar output becomes the value at position (i, j) in the output feature map.

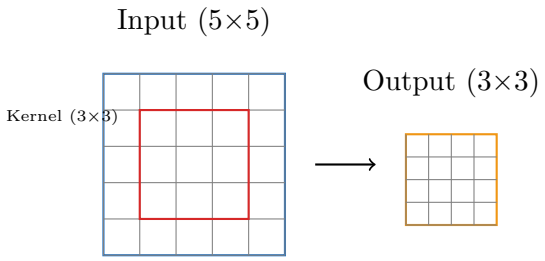

Figure 1: 2D convolution: A 3×3 kernel slides over a 5×5 input, producing a 3×3 output. Each output element is computed by element-wise multiplication and summation over the kernel region.

2.2 Multi-Channel Convolution

Real images have multiple channels (RGB color or deeper features). The convolution operation naturally extends to multi-channel inputs: the kernel becomes a 3D tensor with spatial dimensions and a channel dimension. Convolution is applied to each input channel independently, then results are summed across channels, producing a single scalar per spatial location.

Formally, a filter processing C input channels has shape (height, width, channels). Convolution at each spatial position sums across all channels after element-wise multiplication.

2.3 Multiple Filters and Output Channels

To learn diverse features, we apply multiple filters simultaneously. Each filter produces one output channel. With F filters, the layer output has F channels. These output channels can be interpreted as a learned feature representation of the input.

The number of learnable parameters in a convolutional layer depends only on kernel size, input channels, and output channels, not on spatial dimensions. This parameter efficiency compared to fully connected layers is fundamental to CNN success.

3 Spatial Dimension Considerations

3.1 Output Dimension Calculation

When a kernel of size k is applied to an input of spatial dimension n with no padding and stride 1, the output dimension is reduced:

$$\text{output dimension} = n - k + 1 \tag{1}$$

This reduction occurs because the kernel must fit entirely within the input; at boundaries, there is insufficient context for full kernel placement.

For multi-step strides, where the kernel shifts by s positions per step, the output dimension becomes:

$$\text{output dimension} = \left\lfloor \frac{n-k}{s} \right\rfloor + 1 \tag{2}$$

3.2 Padding Strategies

Spatial dimension reduction presents challenges: information loss at boundaries and rapid shrinking through layers. Padding addresses this by adding context around the input boundary.

Zero Padding adds zero-valued elements around the input boundary. With padding p on all sides:

$$\text{padded dimension} = n + 2p \tag{3}$$

The output dimension with padding is then:

$$\text{output dimension} = \frac{n + 2p - k}{s} + 1 \tag{4}$$

Same Padding uses sufficient padding such that output dimension equals input dimension (assuming unit stride). For a kernel of size k, this requires:

$$p = \frac{k - 1}{2} \tag{5}$$

Valid Padding uses no padding ($p = 0$), allowing spatial dimension reduction.

3.3 Stride and Receptive Field Growth

Using stride $s > 1$ reduces output spatial dimensions by factor s. This reduces computation and memory but may discard spatial information. A stride-2 layer processes every other position, halving output resolution.

Receptive field is the region of input that influences a given output and grows with kernel size and stride. A receptive field is larger at deeper layers, with early layers focusing on local details and later layers integrating global context.

Algorithm 1 Computing Receptive Field Size

```
Initialize RF = 1 at input
for each convolutional layer with kernel size k and stride s do
    RF = RF + (k − 1) × stride_product
end for
```

4 Filters and Feature Learning

4.1 Low-Level Feature Detection

In early layers, filters learn to detect simple patterns: edges, corners, and textures. An edge detector filter might have positive weights on one side and negative on the opposite, producing strong responses at intensity boundaries.

These low-level features emerge from training through gradient-based optimization, without explicit programming. The network discovers that these features are useful for the task.

4.2 Hierarchical Feature Composition

As data progresses through layers, features become increasingly abstract. Early layer outputs (edge, texture) become inputs to deeper layers, which combine them to detect object parts (eyes, wheels). Even deeper layers combine parts to recognize objects.

This hierarchical composition occurs naturally through the stacked architecture: each layer receives features from the previous layer, allowing progressively higher-level pattern recognition.

4.3 Visualizing Learned Filters

Examining learned filters provides insight into network function. Early layer filters resemble edge detectors and oriented patterns. Later layers show more complex patterns representing object parts. This empirically validates the hierarchical representation hypothesis.

5 Pooling Operations

5.1 Spatial Dimension Reduction

Pooling layers reduce spatial dimensions without learning parameters. They summarize local regions with a single value. Two common pooling operations are:

Max Pooling: Divides the feature map into non-overlapping regions and outputs the maximum value in each region. For 2×2 pooling with stride 2, a 4×4 input becomes 2×2 output.

Average Pooling: Outputs the mean of each region instead of maximum. This is less commonly used but provides a smoother summarization.

5.2 Benefits of Pooling

Pooling provides multiple benefits:

1. **Computational Efficiency:** Reduces spatial dimensions and subsequent computation exponentially

2. **Noise Robustness:** Summarization reduces sensitivity to small input perturbations

3. **Translation Invariance:** Small spatial shifts in input produce similar pooled outputs

4. **Hierarchical Aggregation:** Combines information from local regions into broader context

5.3 Global Average Pooling

Global average pooling computes the mean across all spatial locations of a feature map, producing a single value per channel. This dramatically reduces spatial dimensions to zero while aggregating information globally.

Global average pooling is often used before fully connected classification layers, converting spatial feature maps into vector representations suitable for classification.

6 Backpropagation Through Convolutional Layers

6.1 Forward Pass Storage

Backpropagation requires access to forward pass activations. For convolutional layers, this includes all pre-activation and post-activation values across spatial locations and channels.

6.2 Gradient Computation for Filters

The gradient of loss with respect to a filter element is computed by considering contributions from all spatial locations where that filter was applied. For each filter position, the gradient is computed by element-wise multiplication of the error signal with the input region.

The gradients are accumulated across all spatial positions, contributing to the final filter gradient.

6.3 Error Signal Propagation

The error signal must be propagated backward through both the convolution operation and activation functions. The spatial structure of convolution creates dependencies: each output depends on multiple input positions.

The error at each input position must be computed by summing contributions from all filter positions that involved that input. This is equivalent to applying a transposed convolution of the error signal with the filter.

7 Advanced Convolution Variants

7.1 1×1 Convolutions

Convolutions with 1×1 kernel have no spatial extent—they operate at each spatial location independently, equivalent to applying a fully connected layer at each location.

Despite this simplicity, 1×1 convolutions serve important functions:

1. Dimensionality reduction (bottleneck layers reducing channels before expensive operations)
2. Channel mixing (learning combinations of input channels)
3. Nonlinearity injection (adding activation functions between layers)

7.2 Dilated Convolutions

Dilated convolution inserts gaps (dilation factor d) between kernel elements. A 3×3 kernel with dilation 2 effectively has receptive field 5×5 but uses only 9 parameters instead of 25.

Dilated convolutions are useful for:

1. Multi-scale processing (different dilation factors process different scales)
2. Increasing receptive field without adding parameters or computation
3. Semantic segmentation where global context is important

8 Practical CNN Implementation

8.1 Standard Architecture Pattern

Modern CNNs follow a common pattern: repeated blocks of convolution, activation, optional normalization, and pooling. These blocks progressively reduce spatial dimensions while increasing channel count.

```
import torch.nn as nn

class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.conv = nn.Conv2d(in_channels, out_channels,
                              kernel_size=3, padding=1)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = self.relu(x)
        return x

class SimpleCNN(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()
        self.block1 = ConvBlock(3, 32)
        self.pool1 = nn.MaxPool2d(2, 2)

        self.block2 = ConvBlock(32, 64)
        self.pool2 = nn.MaxPool2d(2, 2)

        self.gap = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(64, num_classes)

    def forward(self, x):
        x = self.block1(x)
        x = self.pool1(x)

        x = self.block2(x)
        x = self.pool2(x)

        x = self.gap(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

9 Conclusion

Convolutional neural networks achieve their power through exploiting image structure. The convolution operation provides parameter efficiency through weight sharing while preserving spatial relationships. Pooling reduces spatial dimensions while extracting salient features. The hierarchical composition of convolutional blocks enables learning of in-

creasingly abstract features. Understanding the convolution operation, spatial dimension management, and advanced variants provides foundation for designing effective architectures.

References

- LeCun, Y., Bottou, L., Bengio, Y., Haffner, P. (1998). Gradient-based learning applied to document recognition. PIEEE.
- Dumoulin, V., Visin, F. (2016). A guide to convolution arithmetic for deep learning. arXiv:1603.07285.
- Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. CVPR.
- He, K., Zhang, X., Ren, S., Sun, J. (2015). Spatial pyramid pooling in deep CNNs for visual recognition. TPAMI.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep learning. MIT press.