Convolutional Neural Networks

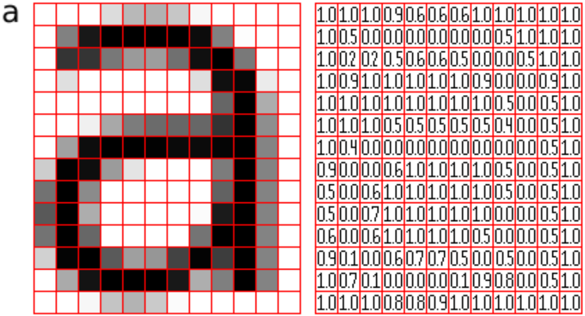
Zichao Yang

Zhongnan University of Economics & Law

Date: December 27, 2025

Convolutional Neural Networks

Convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples: (1) time-series data (1-D grid), (2) image data (2-D or 3-D grid).

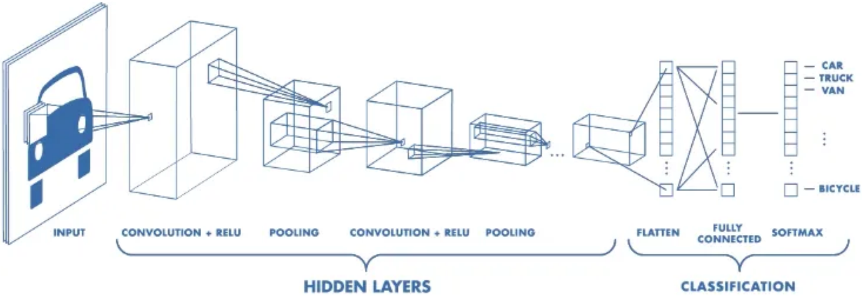


1.0	1.0	1.0	0.9	0.6	0.6	0.6	1.0	1.0	1.0	1.0	1.0
1.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	1.0	1.0
1.0	0.2	0.2	0.5	0.6	0.6	0.5	0.0	0.0	0.5	1.0	1.0
1.0	0.9	1.0	1.0	1.0	1.0	1.0	0.9	0.0	0.0	0.9	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
1.0	1.0	1.0	0.5	0.5	0.5	0.5	0.5	0.4	0.0	0.5	1.0
1.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0
0.9	0.0	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.6	1.0	1.0	1.0	1.0	1.0	0.5	0.0	0.5	1.0
0.5	0.0	0.7	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.5	1.0
0.6	0.0	0.6	1.0	1.0	1.0	1.0	0.5	0.0	0.0	0.5	1.0
0.9	0.1	0.0	0.6	0.7	0.7	0.5	0.0	0.5	0.0	0.5	1.0
1.0	0.7	0.1	0.0	0.0	0.0	0.1	0.9	0.8	0.0	0.5	1.0
1.0	1.0	1.0	0.8	0.8	0.9	1.0	1.0	1.0	1.0	1.0	1.0

Figure 1: Representation of image as a grid of pixels

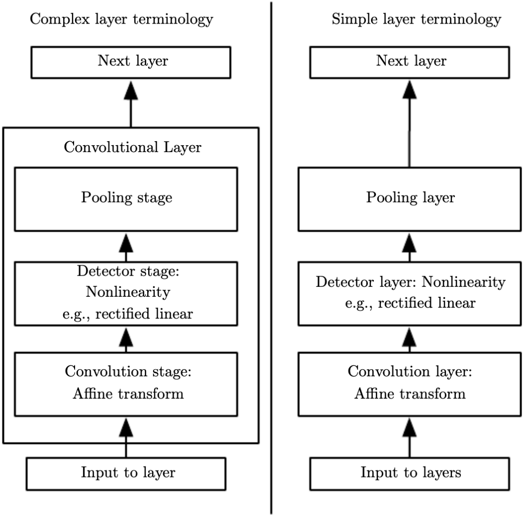
Source: Toward Data Science-Convolutional Neural Networks, Explained

Architecture of a CNN



Source: Toward Data Science-Convolutional Neural Networks, Explained

Components of a Convolutional Layer



Source: Goodfellow, et al. *Deep learning.* MIT press, 2016.

Architecture of a CNN

Part I: Convolutional Layer

The Convolution Operation

First, let's see where the term **convolution** comes from.

In mathematics, convolution is a mathematical operation on two functions (f and g) that produces a third function ($f * g$). The term *convolution* refers to both the result function and to the process of computing it.

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

Convolution is **commutative**, meaning we can equivalently write it as:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(t - \tau) g(\tau) d\tau$$

The Convolution Operation

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

In convolution terminology, the first argument ($f(.)$) is often referred to as the **input**, and the second argument ($g(.)$) as the **kernel** (also called **filter**). The output is sometimes referred to as the **feature map**.

In machine learning applications, we usually deal with **discrete convolution**:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

The Convolution Operation

Meanwhile, the **input** is usually a multidimensional array of **data**, and the **kernel** (or **filter**) is a multidimensional array of **weights** that are learned by the algorithm.

In reality, the input and kernel have to be stored in the memory, so we assume the functions $f(.)$ and $g(.)$ are zero everywhere but in the finite set of points for which store the value. Hence, the value range can be reduced from $(-\infty, \infty)$ to finite.

Finally, we often use convolutions over more than one axis at a time. For example, we use a 2-D image I as our input, and a 2-D kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Convolution VS. Cross-correlation

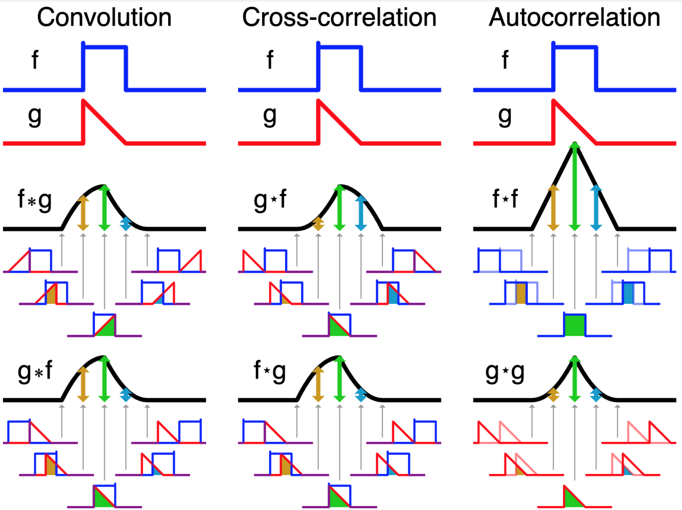
Convolution is a mathematical operation that combines two functions to produce a third function, expressing how the shape of one is modified by the other.

Cross-correlation measures the similarity between two sequences (or functions) as one sequence slides over the other.

$$\begin{aligned} \text{Convolution:} \quad & (f * g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t-\tau) d\tau \\ \text{Cross-correlation:} \quad & (f \star g)(t) := \int_{-\infty}^{\infty} f(\tau) g(t+\tau) d\tau \end{aligned}$$

In cross-correlation, we do not reflect $g(\tau)$ about the y-axis, meaning no need to change $g(\tau)$ to $g(-\tau)$ then perform the integral.

Convolution VS. Cross-correlation



Source: Wikipedia

Convolution VS. Cross-correlation

In the above image example, we have:

Convolution:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Based on **commutativity**, the above convolution equals to:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Cross-correlation:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Many machine learning libraries implement cross-correlation but call it convolution.

Implementation of Convolution

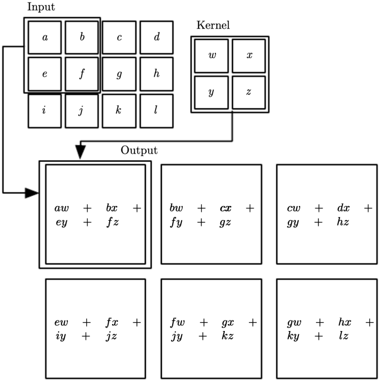


Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

Source: Goodfellow, et al. *Deep learning.* MIT press, 2016.

Sparse Interactions

Convolution leverages three important ideas that can help improve a machine learning system: **sparse interactions**, **parameter sharing** and **equivariant representations**.

In feedforward neural networks, every output unit interacts with every input unit.

Convolutional networks, however, typically have **sparse interactions** (also referred to as sparse connectivity or sparse weights). This is accomplished by making **the kernel smaller than the input**.

This means that we need to store fewer parameters, which: (1) reduce the memory requirements of the model; (2) improve model's statistical efficiency; (3) require fewer operations to compute the output.

Sparse Interactions

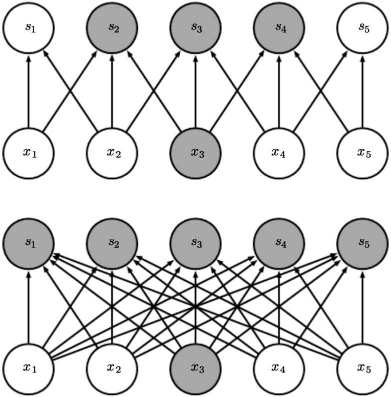


Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, x_3, and also highlight the output units in $\boldsymbol{s}$ that are affected by this unit. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by x_3.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Parameter Sharing

Parameter sharing refers to using the same parameter for more than one function in a model.

In a feedforward neural network, each element of the weight matrix is used exactly once when computing the output of a layer, and the weight is never used again.

In a convolutional neural network, each member of the kernel is used at every position of the input (except perhaps some boundary positions). For every location, the algorithm apply the same set of weights.

Parameter sharing does not reduce the runtime of forward propagation, but it does further **reduce the storage requirements** of the model to the size of kernel K.

Parameter Sharing

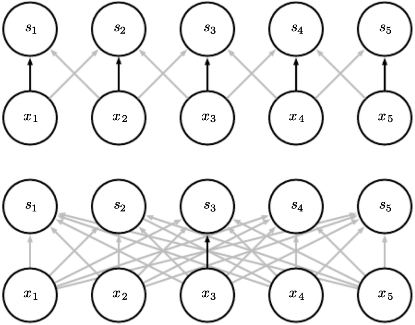


Figure 9.5: Parameter sharing: Black arrows indicate the connections that use a particular parameter in two different models. *(Top)*The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. *(Bottom)*The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

Source: Goodfellow, et al. *Deep learning.* MIT press, 2016.

Equivariant Representations

We say a function $f(x)$ is **equivariant** to a function g if $f(g(x)) = g(f(x))$. Why is the property useful?

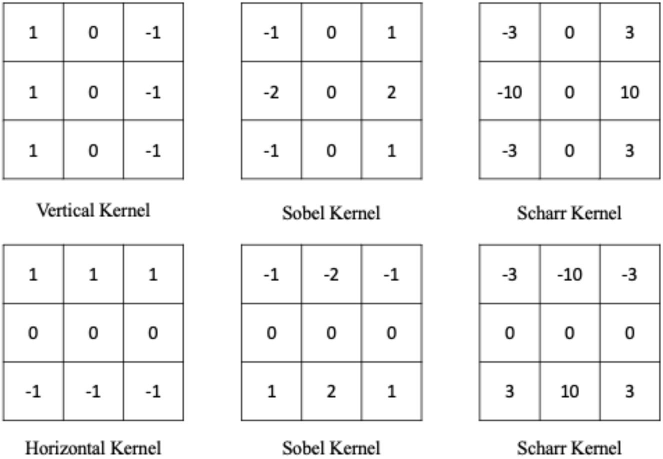
Suppose a convolution layer creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output.

Hence, when we know that some function of a small number of neighboring pixels is useful, we are certain that these functions can be applied to multiple other input locations. For example, if we want to detect edges, the same edges appear everywhere in the image, so it is practical to share parameters across the entire image.

Note: convolution is not naturally equivariant to certain transformations, such as changes in the scale or rotation of an image.

Some Special Kernel

In most cases, the weights in kernels are learned from data via backpropagation, but there are some famous pre-determined kernels:



1	0	-1
1	0	-1
1	0	-1

Vertical Kernel

-1	0	1
-2	0	2
-1	0	1

Sobel Kernel

-3	0	3
-10	0	10
-3	0	3

Scharr Kernel

1	1	1
0	0	0
-1	-1	-1

Horizontal Kernel

-1	-2	-1
0	0	0
1	2	1

Sobel Kernel

-3	-10	-3
0	0	0
3	10	3

Scharr Kernel

Some Special Kernel

Vertical (Horizontal) Kernel: designed to detect vertical (horizontal) edges in an image by detecting in the horizontal (vertical) direction.

Sobel Kernel: a popular edge-detection filter that is used to find the gradient of an image's intensity at each point.

Scharr Kernel: another edge-detection filter that is designed to reduce the impact of noise and to provide a better approximation to the gradient magnitude than the Sobel kernel.

These pre-determined kernels are not learned from data, but they can be applied to images as a pre-processing step to enhance edges and other features before feeding the images into a CNN.

Zero Padding

Zero-padding refers to surrounding the input array with zeroes. Here are the primary functions of zero padding:

- control over output size. Zero padding the input allows us to control the kernel width and the size of the output independently.
- maintain border information. Without padding, the border pixels would be less frequently visited by the kernel, leading to potential loss of important edge information.
- unify varying input sizes.

Stride

In CNNs, **stride** refers to the number of pixels by which the kernel (or filter) moves across the input image. It is an important hyperparameter that controls the spatial dimensions of the output feature map.

Stride = 1: the most common setting where the filter moves one pixel at a time. The kernel is applied at every possible location in the input image.

Stride > 1: kernel skips over multiple pixels during its application. This operation reduces the spatial dimensions of the output feature map.

Stride

The size of the output feature map can be calculated using the following formula:

$$\text{Output Size} = \lfloor \frac{\text{Input Size} - \text{Kernel Size} + 2 * \text{Padding Size}}{\text{Stride}} \rfloor + 1$$

Where:

- Input Size is the height or width of the input image
- Filter Size is the height or width of the kernel
- Padding Size is the number of pixels added to the border of the input image
- Stride is the step size of the kernel
- $\lfloor \cdot \rfloor$ denotes the floor function, which rounds down to the nearest integer

Different Convolutions

Suppose the input image has width of m and the kernel has width of k.

- **Valid Convolution**: no zero-padding is used whatsoever, and the convolution kernel is only allowed to visit positions where the entire kernel is contained entirely within the image. The size of the output is $m - k + 1$.
- **Same Convolution**: just enough zero-padding is added to keep the size of the output equal to the size of the input, m.
- **Full Convolution**: enough zero-padding is added for every pixel to be visited k times in each direction, resulting in an output of size $m + k - 1$.
- **Unshared Convolution**: aka. locally connected layers.
- **Tiled Convolution**: a compromise between a convolutional layer and a locally connected layer.

Different Convolutions

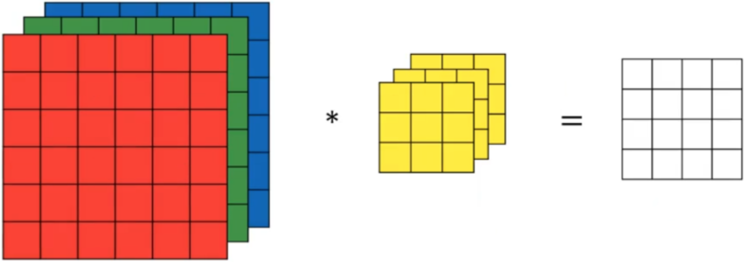
Valid Convolution: all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer.

Same Convolution: the network can contain as many convolutional layers as the available hardware can support. However, the input pixels near the border influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model.

Full Convolution: the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult to learn a single kernel that performs well at all positions in the convolutional feature map.

Convolution Over Volume

Convolution over volume refers to the process of applying a kernel to a multi-channel input, such as a color image or a multi-channel feature map from a previous layer. This operation extends the idea of convolution from 2D (height and width) to 3D (height, width, and depth).



Source: Andrew Ng-Machine Learning Specialization

Architecture of a CNN

Part II: Detector Layer

Detector Layer

Since convolution is a linear operation and images are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity, and we call it **detector layer**.

There are several types of non-linear operations, the popular ones being:

- Sigmoid
- Tanh
- ReLU

Architecture of a CNN

Part III: Pooling Layer

Pooling Layer

Remember: Pooling layers do not require learnable parameters.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. Common pooling functions: **max pooling** and **average pooling**.

Pooling helps to make the representation become approximately invariant to small translations of the input. *Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.*

Pooling Layer

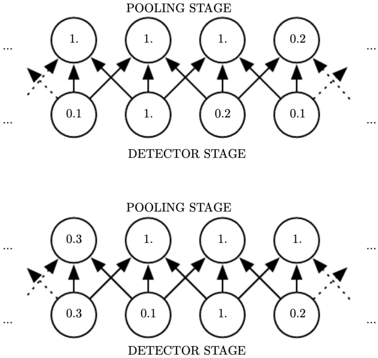


Figure 9.8: Max pooling introduces invariance. *(Top)*A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of one pixel between pooling regions and a pooling region width of three pixels. *(Bottom)*A view of the same network, after the input has been shifted to the right by one pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Pooling Layer

Because pooling summarizes the responses over a whole neighborhood, we can use the pooling layer to reduce the input size of the next layer by setting the **stride** larger than 1, and this can improve the computational efficiency.

For many tasks, pooling is essential for handling inputs of varying size.

For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of the stride between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size.

CNN Architecture Examples

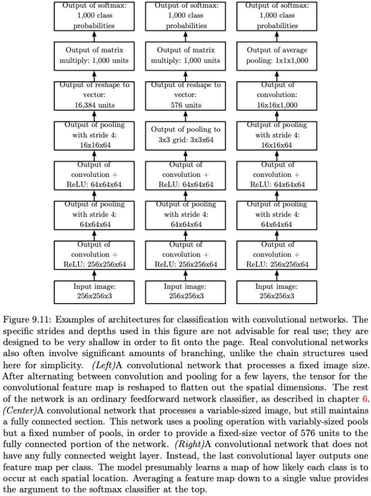

Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 16,384 units
Output of pooling with stride 4: 16x16x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of matrix multiply: 1,000 units
Output of reshape to vector: 576 units
Output of pooling to 3x3 grid: 3x3x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3
Output of softmax: 1,000 class probabilities
Output of average pooling: 1x1x1,000
Output of convolution: 16x16x1,000
Output of pooling with stride 4: 16x16x64
Output of convolution + ReLU: 64x64x64
Output of pooling with stride 4: 64x64x64
Output of convolution + ReLU: 256x256x64
Input image: 256x256x3

Figure 9.11: Examples of architectures for classification with convolutional networks. The specific strides and depths used in this figure are not advisable for real use; they are designed to be very shallow in order to fit onto the page. Real convolutional networks also often involve significant amounts of branching, unlike the chain structures used here for simplicity. *(Left)*A convolutional network that processes a fixed image size. After alternating between convolution and pooling for a few layers, the tensor for the convolutional feature map is reshaped to flatten out the spatial dimensions. The rest of the network is an ordinary feedforward network classifier, as described in chapter 6. *(Center)*A convolutional network that processes a variable-sized image, but still maintains a fully connected section. This network uses a pooling operation with variably-sized pools but a fixed number of pools, in order to provide a fixed-size vector of 576 units to the fully connected portion of the network. *(Right)*A convolutional network that does not have any fully connected weight layer. Instead, the last convolutional layer outputs one feature map per class. The model presumably learns a map of how likely each class is to occur at each spatial location. Averaging a feature map down to a single value provides the argument to the softmax classifier at the top.

Source: Goodfellow, et al. *Deep learning*. MIT press, 2016.

Prior Assumptions in Convolution and Pooling

We can imagine a convolutional net as being similar to a fully connected net, but with an infinitely strong prior over its weights.

Prior assumptions in the convolution layer:

- the weights for one hidden unit must be identical to the weights of its neighbor, but shifted in space
- the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit

Prior assumption in the pooling layer:

- each unit should be invariant to small translations

Takeaway: convolution and pooling can cause **underfitting**. If a task relies on preserving precise spatial information, then using pooling on all features can increase the training error.

CNN in PyTorch

User Guide: https://pytorch.org/docs/stable/nn.html

```
import torch

torch.nn.Conv2d(in_channels, out_channels, kernel_size,
               stride=1, padding=0,  padding_mode='zeros')
torch.nn.Tanh()
torch.nn.MaxPool2d(kernel_size)
```

Let’s code out a CNN model using PyTorch!