Deep Learning (3): Convolutional Neural Networks
(for high school and first-year university students)

Networks for data with spatial structure

Planned and edited by Yoshio Okamoto / Written by Claude Opus 5.5 (Anthropic)

Ver. 0.2 October 7, 2026

Note: The following content was generated by AI. The editor has checked it, but please be aware that it may still contain errors or mistaken assumptions.

© 2026 Yoshio Okamoto. Free to use for educational purposes (please cite the source).

Introduction

The networks in Parts 1 and 2 were **fully connected**: every component of the input was connected to every unit of the next layer. Part 3 deals with **convolutional neural networks** (CNNs), which are suited to data **in which the spatial arrangement has meaning**, such as images.
When we look at a photo, what we first notice is **what nearby parts look like**, such as "there is a line here" or "the color changes here," and by combining these we recognize "this is a cat's face." A CNN builds this way of "looking nearby and combining" into the shape of the network.

Part	Title
Part 1	Basics of Neural Networks
Part 2	Backpropagation and Training in Practice
Part 3	**Convolutional Neural Networks**
Part 4	Sequence Data and Transformers

How to read this booklet The main text is written so that it can be read with high-school mathematics (the summation sign $\sum$ and simple arithmetic). The mathematical properties of convolution (the difference from cross-correlation, a proof of the property under translation, the backpropagation formula) and the general formula with multiple channels are collected in the **footnotes** and in **Appendix A** at the end. You can follow the main text even if you skip the appendix. All numerical values in the examples were actually computed (program in Appendix B).

Notation

- One-dimensional data (a signal) is written $x_0, x_1, \ldots, x_{n-1}$, and two-dimensional data (an image) X_{ij} (the pixel in row i, column j). Indices are counted from 0.
- A filter (weights) is written w_u (one-dimensional) or W_{uv} (two-dimensional), and its size is k ($k \times k$ in two dimensions).
- The size of an image is written "height × width × number of channels." **Channels** are several values stacked at the same position, like the red, green and blue of a color image.

1 Problems with fully connected networks

1.1 Too many parameters

A color image of 224×224 pixels has three channels, red, green and blue, so it is $224 \times 224 \times 3 = 150{,}528$ numbers. Connecting it to a fully connected hidden layer (1000 units) needs

$$150{,}528 \times 1000 \approx 1.5 \times 10^8$$

weights alone. The first layer alone has 150 million parameters, so training needs an enormous amount of data and overfitting happens easily.

> Full connection is like each of 1000 inspectors looking at **all** 150,000 pixels to make a judgment. Moreover, each inspector has a different way of looking (different weights), so there is an enormous amount to memorize.

1.2 It does not use the properties of images

A fully connected network handles the input pixels lined up in a single row, so it does not use the information about **which pixels are next to which**. Shuffling the order of the pixels gives the same problem as far as the network is concerned. Even if you cut a photo up like a jigsaw puzzle and rearrange the pieces, a fully connected network does not care.

But images have the following properties.

1. **Locality**: meaningful features (edges, corners, textures) are determined by the relations between nearby pixels.
2. **Independence of position**: a cat's eye is a cat's eye whether it is at the top left or the bottom right of the image.

> A convolutional neural network builds these two properties into its structure.

2 One-dimensional convolution

2.1 Definition

First consider one dimension. The **convolution** of a signal x_i of length n and a filter w_u of length k is defined as[1]

$$y_i = \sum_{u=0}^{k-1} w_u \, x_{i+u} + b, \qquad i = 0, 1, \ldots, n-k.$$

For example, with $k = 3$,

$$y_0 = w_0 x_0 + w_1 x_1 + w_2 x_2 + b, \qquad y_1 = w_0 x_1 + w_1 x_2 + w_2 x_3 + b, \qquad \ldots$$

> It is the operation of sliding the filter over the signal one step at a time and taking **the sum of the products of the overlapping parts**. The length of the output is $n - k + 1$.
> Imagine sliding a long, narrow **peephole (a ruler)** over the signal from one end to the other, multiplying the few values seen through the window by fixed weights, adding them, and recording the result at each position. You may also think of the filter as **a template** expressing "what shape it is looking for."

[1] The "convolution" of mathematics and signal processing is the operation in which the filter is flipped left to right before overlapping; strictly speaking, the definition here is called "cross-correlation." In deep learning there is no difference between the two, so by custom the unflipped form is called "convolution" (Appendix A.1).

Compared with full connection, convolution can be seen as a special fully connected layer with the following two constraints (Figure 1).

- **Local connection**: each output is connected only to k nearby inputs.
- **Weight sharing**: the outputs at every position use the same k weights $w_0, \ldots, w_{k-1}$.

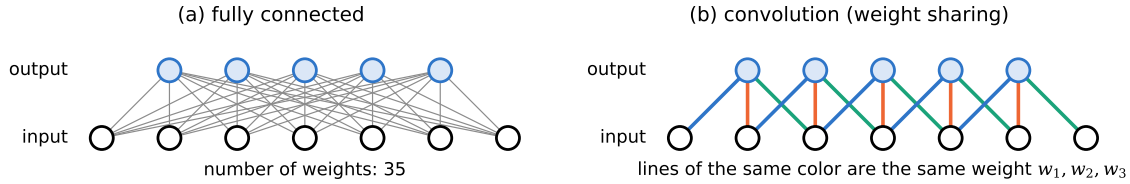


Figure 1 Comparison of (a) full connection and (b) convolution (7 inputs, 5 outputs, filter length 3). In full connection, all 35 weights are different; in convolution, 3 weights are reused at every position.

> In the inspector analogy above, with convolution each inspector looks at **only a few pixels in their own neighborhood** (local connection), and moreover **everyone judges by the same manual** (the same weights; weight sharing). There is much less to memorize.

2.2 Example: smoothing and differences

We apply two kinds of filters to a signal of 8 values with a step, $x = (0, 0, 0, 0, 3, 3, 3, 3)$ (program in Appendix B).

Filter	w	Output y
Moving average	$(1/3,\ 1/3,\ 1/3)$	$(0,\ 0,\ 1,\ 2,\ 3,\ 3)$
Difference	$(-1,\ 0,\ 1)$	$(0,\ 0,\ 3,\ 3,\ 0,\ 0)$

For example, y_2 of the moving average is $\frac{1}{3}(x_2 + x_3 + x_4) = \frac{1}{3}(0 + 0 + 3) = 1$, and y_2 of the difference is $-x_2 + 0 \cdot x_3 + x_4 = -0 + 3 = 3$.

- The **moving-average** filter averages three neighboring values and smooths out sudden changes. It is the same as taking a "7-day moving average" of daily temperatures to smooth out small ups and downs.
- The **difference** filter computes $y_i = x_{i+2} - x_i$ and outputs **large values only where the value is changing**. It is a detector of steps (edges).

For a noisy signal, the result is as in Figure 2. Applying the difference filter directly also emphasizes the noise, but taking differences after smoothing the noise with a moving average makes the position of the step stand out clearly.

> Operations like these have long been used in signal processing as **digital filters** (FIR filters). Removing noise from seismograph records, or looking for the moment a wave arrives, are computations of the same kind. What is new about CNNs is that **the filter coefficients are not designed by people but learned from data**.

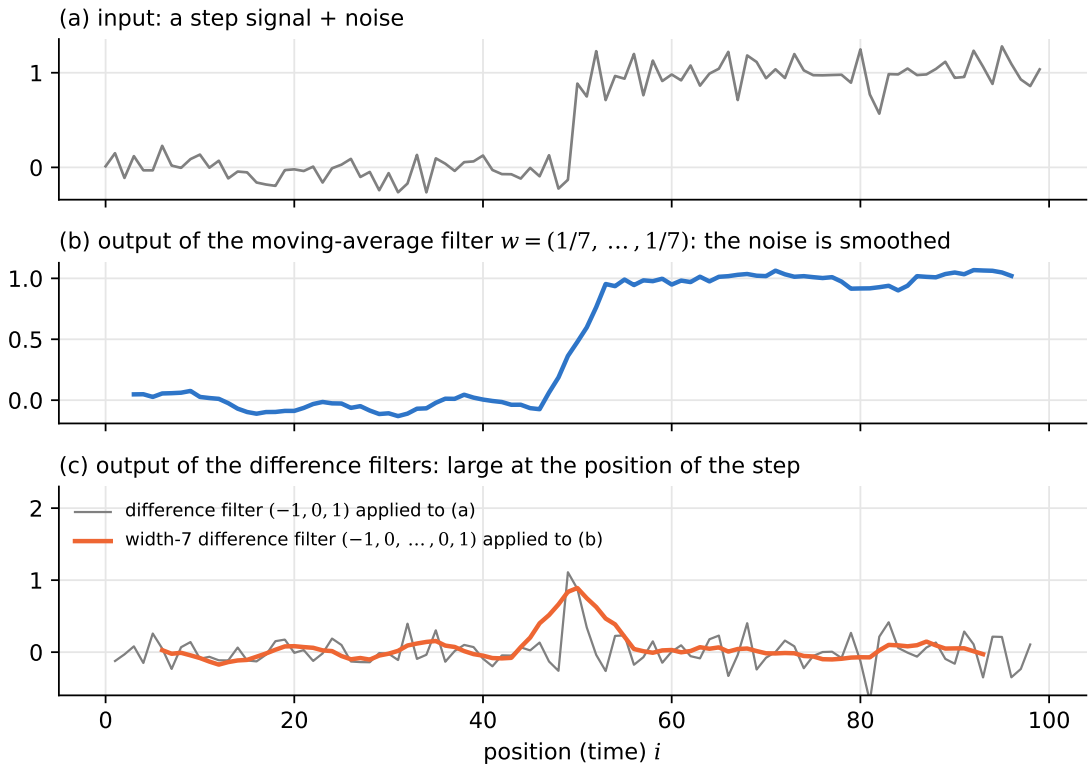


Figure 2 Examples of one-dimensional convolution. (a) A signal with a step at position 50, with noise added. (b) The 7-point moving average. (c) Outputs of difference filters (gray: applied directly to (a); orange: a width-7 difference filter applied to (b)).

3 Two-dimensional convolution

3.1 Definition

The convolution of an image X and a $k \times k$ filter W is

$$Y_{ij} = \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} W_{uv}\, X_{i+u,\, j+v} + b.$$

There are two $\sum$ signs because we add over both the vertical (u) and horizontal (v) directions of the filter; a 3×3 filter gives a sum of 9 products.

> We slide a $k \times k$ window over the image vertically and horizontally and take the sum of the products of the overlapping parts (Figure 3). It is the work of **going over a photo with a magnifying glass, little by little from top left to bottom right**, and writing down at each place a score for "how similar it is to the shape we are looking for."

The filter of Figure 3,

$$W = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix},$$

"subtracts the left column from the right column" and detects **vertical edges**. In the output, only the columns at the boundary between 0 and 1 in the input become 3, and uniform parts become 0. The filter with its rows and columns swapped ("subtract the top row from the bottom

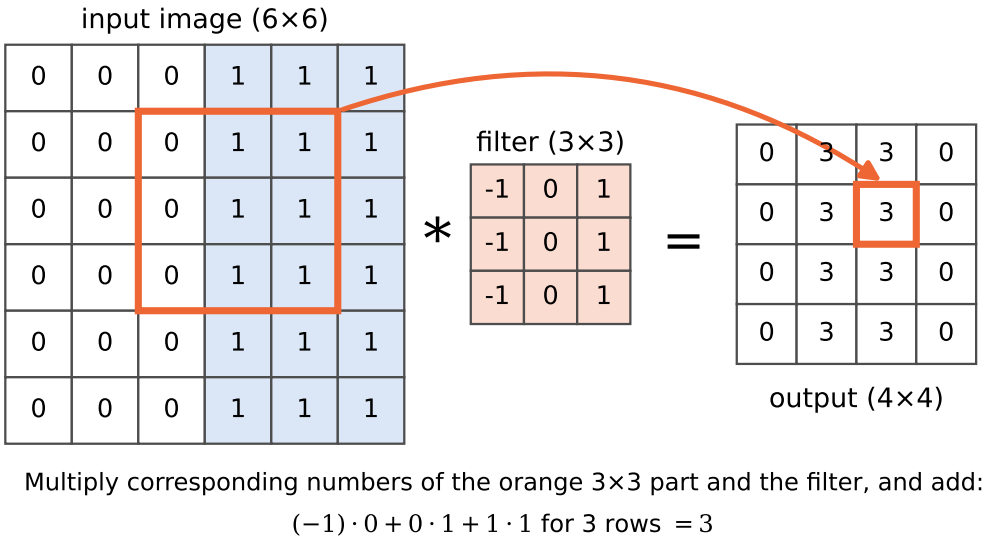


Figure 3 Computing a two-dimensional convolution. Applying a 3×3 filter to a 6×6 input image (right half is 1) gives a 4×4 output. The computation over the orange-framed part gives the value 3 in the orange frame of the output.

row")[2] detects **horizontal edges**, and its output for the input of Figure 3 (which has only a vertical edge) is all 0.

Figure 4 is an example of applying these filters to a slightly more complex image.

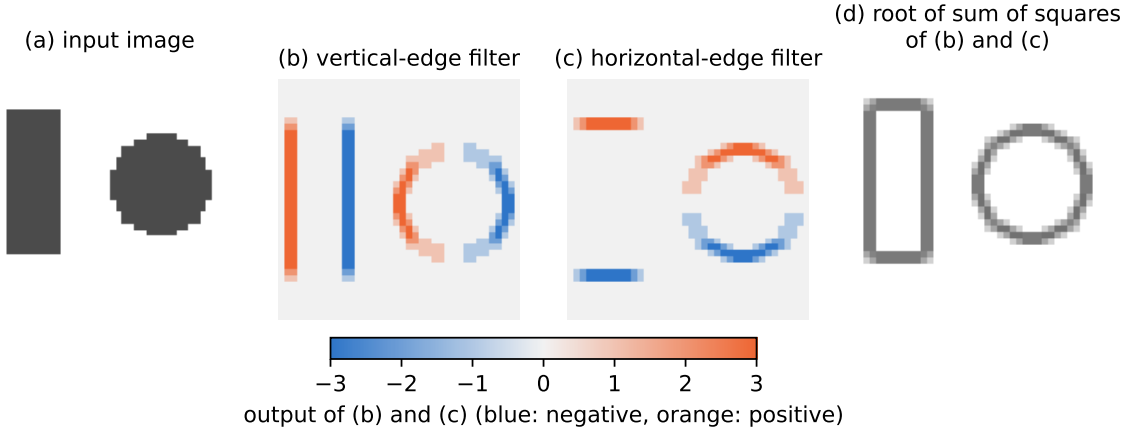


Figure 4 Examples of edge-detecting filters (the input is 40×40 pixels). Outputs of (b) the vertical-edge filter and (c) the horizontal-edge filter (rows and columns swapped). Edges where brightness increases are orange (positive), and edges where it decreases are blue (negative). (d) is the strength of the edge combining the two.

Edge detection is a basic operation of image processing, used for example to extract coastlines or the edges of clouds from satellite images.

3.2 Output size: padding and stride

- **Padding**: fill a border of width p around the input with 0 before convolving. It is like putting **a frame** around a photo, so that the computation can be done even when the

[2] Swapping the rows and columns of a matrix is called **transposing**, written W^{T}.

window sticks out past the edge of the image. With $k = 3$, choosing $p = 1$ makes the output the same size as the input.

- **Stride**: slide the window by s pixels instead of 1. Like **crossing stepping stones while skipping every other one**, $s = 2$ makes the height and width of the output about half.

When the input size (the number of pixels vertically or horizontally) is n, the output size is

$$(\text{the integer part of } \tfrac{n+2p-k}{s}) \ + 1.$$

Padding makes the width $n + 2p$, and we count the number of places where a window of size k can be put. In Figure 3, $n = 6$, $k = 3$, $p = 0$, $s = 1$, so $6 - 3 + 1 = 4$.

3.3 Multiple channels

In real CNNs, both input and output have multiple channels.

- When the input has C_{in} channels, one filter has size $k \times k \times C_{\text{in}}$ and takes the sum of products over all channels (for a color image, the sums of products over the $k \times k$ windows of red, green and blue are added together over the 3 channels).
- Preparing C_{out} such filters gives an output with C_{out} channels.

> It is easy to think of channels as **several maps of the same area stacked on top of each other**. Just as weather maps include maps of temperature, pressure and humidity, a color image has three "maps," red, green and blue, stacked together. A filter looks at these stacked maps at the same time and makes one new map (a feature map).

Written as a formula, quantities with four indices appear (Appendix A.2). Each filter has $k \times k \times C_{\text{in}}$ weights and one bias, so the number of parameters is

$$k \times k \times C_{\text{in}} \times C_{\text{out}} + C_{\text{out}}.$$

For example, applying 64 filters of size 3×3 to a color image ($C_{\text{in}} = 3$) gives $3 \times 3 \times 3 \times 64 + 64 = 1{,}792$.

> Note that **the number of parameters does not depend on the size of the image**. Compared with the 150 million of the fully connected layer in Section 1.1, it is smaller by orders of magnitude.

Each output channel corresponds to a different filter, so it becomes **a map in which a different kind of feature has been detected** (a **feature map**), such as "vertical edges," "horizontal edges" or "a particular color."

4 Properties of convolution

4.1 Weight sharing and translation

> Since the same filter is used at every position, **if the input is shifted, the output shifts by the same amount**.

For example, shifting the step signal of Section 2.2 one place to the right, to $(0, 0, 0, 0, 0, 3, 3, 3)$, shifts the output of the difference filter one place to the right as well, to $(0, 0, 0, 3, 3, 0)$ (a general proof is in Appendix A.3). This property is called **translation equivariance**. Wherever a cat's eye is in the image, the same filter responds there. This builds property 2 of Section 1.2 into the structure.

If you walk along a beach with a metal detector, the detector responds with the same sensitivity everywhere. If a coin is buried 1 m further to the right, the place where it responds is simply 1 m further to the right. This is the effect of using the same filter (the same detector) at every position.

4.2 Receptive field

One pixel of the output of a single 3×3 convolution uses only the information in a 3×3 region of the input. This region is called the **receptive field**. Stacking two 3×3 convolutions widens the receptive field to 5×5, and stacking L of them to $(2L+1) \times (2L+1)$ (each additional layer widens it by one pixel up, down, left and right).

The more layers are stacked, the more the network can represent **features that combine information from a wide area**.
In a neighborhood association, a block leader only needs to watch the few houses of their block, but the association head who collects the reports of the block leaders can see the whole neighborhood, and someone who collects the reports of the association heads can see the whole city. Even if each person sees only a narrow area, by piling up reports the higher levels learn about wider areas.

5 Pooling

Pooling is an operation that divides a feature map into small regions and shrinks it by representing each region with one value. The most commonly used is **max pooling**, which takes the maximum of each region (Figure 5). It is like choosing, from each group, the value that responded most strongly as its representative.

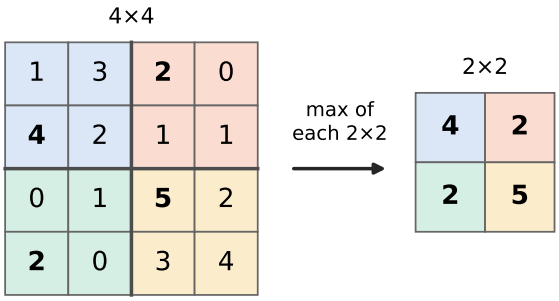


Figure 5 2×2 max pooling (stride 2). A 4×4 feature map becomes 2×2.

Pooling has the following effects.

- The feature map becomes half as tall and half as wide, reducing the computation in later layers.
- The receptive field widens quickly.
- The output changes less when the position of a feature shifts by about one pixel (robustness to small shifts in position).

Pooling has no parameters to learn.

It is like reducing the scale of a map. On a reduced map, you can no longer see the exact positions of small roads, but you can still tell "there is a station around here." **It keeps**

“what is there” while discarding the fine details of “exactly where,” making the whole easier to see.

6 The overall structure of a CNN

6.1 A typical structure

A typical CNN repeats “convolution → ReLU → pooling” several times to extract features, and finally classifies with a fully connected layer and the softmax function (Figure 6).

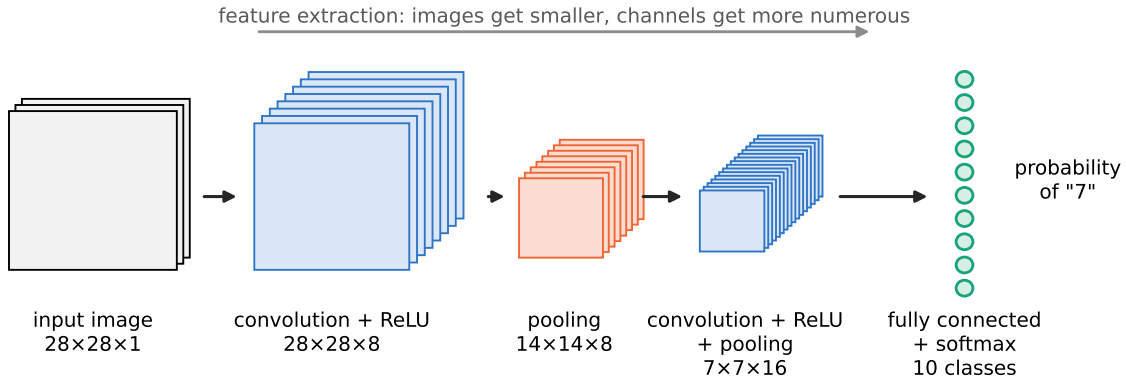


Figure 6 An example structure of a small CNN that classifies handwritten digits (28×28 pixels). The convolution layers use 3×3 filters (padding 1), and pooling is 2×2 max pooling.

The numbers of parameters of the network in Figure 6 are as follows.

Layer	Computation	Parameters
Convolution 1 ($1 \to 8$ channels)	$3 \times 3 \times 1 \times 8 + 8$	80
Convolution 2 ($8 \to 16$ channels)	$3 \times 3 \times 8 \times 16 + 16$	1,168
Fully connected ($7 \times 7 \times 16 = 784 \to 10$)	$784 \times 10 + 10$	7,850
Total		**9,098**

This is about one ninth of the 79,510 of the fully connected network of Part 1 ($784 \to 100 \to 10$).

With fewer parameters (knobs) and a structure that fits the properties of images, high accuracy can be obtained even with little data.

6.2 Training

CNNs are also trained by the backpropagation of Part 2. As we saw in Part 2, the gradient of a weight was “the error that arrived from behind × the value that came into the weight.” In a convolution layer, the same weight is used at every position, so **the gradient of a weight is the sum of these values over all positions** (the formula is in Appendix A.4). It is like collecting the “points to fix” from all the inspectors who work by the same manual, and revising the manual once. In max pooling, the backpropagated error is sent only to the position that had the maximum in each region (the value chosen as representative).

6.3 What is being learned

When trained CNNs are examined, the filters of the first layer often resemble **detectors of edges and color changes**, like those of Figure 4, and it has been observed that deeper layers respond to more complex and abstract features such as **textures, parts (eyes, wheels, etc.) and whole objects**.

> The "hidden layers transforming the representation" that we saw in Part 1 appears in images as **a hierarchy from simple features to complex features**. Lines gather into corners and curves, corners and curves gather into eyes and ears, and eyes and ears gather into a face. Without people designing edge-detecting filters, the filters needed for the purpose of classification are obtained by training.

6.4 History and applications

- LeNet (LeCun et al., 1998), which used a CNN to recognize handwritten digits, is an early representative example.
- In 2012, a large CNN (AlexNet) trained using GPUs far outperformed earlier methods in a competition to classify about 1.2 million images into 1000 categories; this triggered the current boom in deep learning.
- Besides image classification, CNNs are used for detecting the positions of objects, assisting diagnosis from medical images, analyzing satellite images, and more.
- One-dimensional CNNs are also used for speech and various time-series data (for example, detecting the arrival of waves in seismic waveforms).

7 Summary

> 1. For large inputs such as images, fully connected networks have an enormous number of parameters and do not use the spatial arrangement of the pixels.
> 2. Convolution is a special layer with "local connection" and "weight sharing," reusing the same filter at every position. The number of parameters does not depend on the size of the input.
> 3. Filters work as feature extractors, such as smoothing and edge detection, and when the input is shifted, the output shifts by the same amount (translation equivariance).
> 4. Pooling shrinks feature maps and gives robustness to small shifts in position.
> 5. A CNN stacks "convolution → ReLU → pooling" and learns hierarchically from simple features to complex ones. The filters are not designed by people but learned by backpropagation.

Coming next Images were data with a spatial arrangement, but text and time series are data with a **temporal order**. Part 4 deals with RNNs, which handle sequence data, and with the Transformer, the foundation of today's LLMs.

Appendix A The mathematics of convolution

This appendix uses partial derivatives and quantities with many indices.

A.1 "Convolution" and "cross-correlation"

The convolution of mathematics and signal processing is

$$(w * x)_i = \sum_u w_u \, x_{i-u},$$

in which the filter is **flipped** before overlapping ($*$ is the symbol of the convolution operation, not multiplication). The definition in the main text, $\sum_u w_u \, x_{i+u}$, is strictly an operation called **cross-correlation**. In deep learning, the filter w is found by training, so whether it is flipped or not makes no difference except that the learned w comes out flipped. Therefore, by custom, the unflipped form is called "convolution."

A.2 Convolution with multiple channels

When the input has C_{in} channels and the output C_{out} channels, the (i, j) entry of output channel c' is

$$Y_{c',\,i,\,j} = \sum_{c=1}^{C_{\text{in}}} \sum_{u=0}^{k-1} \sum_{v=0}^{k-1} W_{c',\,c,\,u,\,v} \, X_{c,\,i+u,\,j+v} + b_{c'}.$$

W is a **fourth-order tensor** with four indices, and its number of entries, $k \times k \times C_{\text{in}} \times C_{\text{out}}$, is the number of weights.

Output size The output size of Section 3.2 is written with the floor function ($\lfloor a \rfloor$ is the largest integer not exceeding a) as $\left\lfloor \dfrac{n + 2p - k}{s} \right\rfloor + 1$.

A.3 Translation equivariance

For the input shifted m places to the right, $x'_i = x_{i-m}$, the output is

$$y'_i = \sum_u w_u \, x'_{i+u} = \sum_u w_u \, x_{i-m+u} = y_{i-m},$$

which is the original output shifted by m (the bias is omitted).

A.4 Backpropagation through a convolution layer

In one dimension, the weight w_u of the output $y_i = \sum_u w_u \, x_{i+u} + b$ is used in the outputs at all positions i. By "add over all paths" of the chain rule,

$$\frac{\partial L}{\partial w_u} = \sum_i \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial w_u} = \sum_i \frac{\partial L}{\partial y_i} \, x_{i+u}.$$

$\partial L / \partial y_i$ is "the error that arrived from behind at position i," and x_{i+u} is "the value that came into the weight w_u at that position," so this is Part 2's "error $\times$ incoming value" added over all positions.

Appendix B Python code (standard library only)

This program computes the examples of Sections 2.2, 3.1 and 5 of the main text. It uses no external libraries, only the standard features of Python 3.

```
# Convolution and pooling (standard library only)

def conv1d(x, w):
    """1D convolution (no padding, stride 1)"""
    k = len(w)
    return [sum(w[u] * x[i + u] for u in range(k)) for i in range(len(x) - k + 1)]

def conv2d(img, w):
    """2D convolution (no padding, stride 1)"""
    H, W, k = len(img), len(img[0]), len(w)
    return [[sum(w[u][v] * img[i + u][j + v] for u in range(k) for v in range(k))
             for j in range(W - k + 1)] for i in range(H - k + 1)]

def maxpool(a, s=2):
    """s x s max pooling (stride s)"""
    return [[max(a[i + u][j + v] for u in range(s) for v in range(s))
             for j in range(0, len(a[0]), s)] for i in range(0, len(a), s)]

def show(name, a):
    print(name)
    for row in a:
        print("  " + " ".join(f"{v:3g}" for v in row))

# --- 1D: a signal with a step ---
x = [0, 0, 0, 0, 3, 3, 3, 3]
print("input          ", x)
print("moving average ", [round(v, 2) for v in conv1d(x, [1/3, 1/3, 1/3])])
print("difference     ", conv1d(x, [-1, 0, 1]))

# --- 2D: a 6x6 image with a vertical edge ---
img = [[0, 0, 0, 1, 1, 1] for _ in range(6)]
vert = [[-1, 0, 1],
        [-1, 0, 1],
        [-1, 0, 1]]          # filter that detects vertical edges
horiz = [list(r) for r in zip(*vert)]   # swapping rows and columns gives one for horizontal edges
show("input image", img)
show("output of the vertical-edge filter", conv2d(img, vert))
show("output of the horizontal-edge filter", conv2d(img, horiz))

# --- max pooling ---
a = [[1, 3, 2, 0],
     [4, 2, 1, 1],
     [0, 1, 5, 2],
     [2, 0, 3, 4]]
show("before pooling", a)
show("after 2x2 max pooling", maxpool(a))
```

Output:

```
input           [0, 0, 0, 0, 3, 3, 3, 3]
moving average  [0.0, 0.0, 1.0, 2.0, 3.0, 3.0]
difference      [0, 0, 3, 3, 0, 0]
input image
    0   0   0   1   1   1
    0   0   0   1   1   1
    0   0   0   1   1   1
    0   0   0   1   1   1
    0   0   0   1   1   1
    0   0   0   1   1   1
output of the vertical-edge filter
```

```
    0   3   3   0
    0   3   3   0
    0   3   3   0
    0   3   3   0
output of the horizontal-edge filter
    0   0   0   0
    0   0   0   0
    0   0   0   0
    0   0   0   0
before pooling
    1   3   2   0
    4   2   1   1
    0   1   5   2
    2   0   3   4
after 2x2 max pooling
    4   2
    2   5
```