# Fundamentals of CNNs I

Course:
INFO-6152 Deep Learning with Tensorflow & Keras 2

FANSHAWE

Developed by:
Mohammad Noorchenarboo

September 28, 2025

## Current Section

## From ANN to CNN: The Next Step in Deep Learning

We have seen that ANNs face three major problems with image data: 1. **Loss of spatial structure** after flattening. 2. **Explosion in parameters** due to full connections. 3. **Lack of translation invariance** when objects shift positions.

**Definition**

A Convolutional Neural Network (CNN) is a specialized type of neural network designed to process data with a grid-like topology, such as images.
Instead of connecting every input pixel to every neuron, CNNs use:

Local receptive fields + Weight sharing

This allows them to efficiently capture spatial patterns like edges, textures, and shapes.

## From ANN to CNN: The Next Step in Deep Learning

### CNN as a Camera Lens

Think of a CNN like a camera lens that scans across the image. Instead of memorizing every single pixel separately, it looks for patterns (edges, corners, textures) no matter where they appear. This makes it efficient and translation-friendly.

### High-Level Comparison: ANN vs CNN

- **ANN:** Treats images as flat vectors. Cannot naturally preserve spatial structure.
- **CNN:** Processes images as 2D grids. Learns local features and builds them into complex shapes.

### Mistake to Avoid

Do not assume CNNs are just bigger ANNs. They use fundamentally different mechanisms (convolutions, pooling, etc.), which we will explore in the upcoming sections.

## From ANN to CNN: The Next Step in Deep Learning

**Warning**

While CNNs solve many problems of ANNs, they also introduce new design choices (filter size, stride, padding, etc.) that must be carefully tuned.

**Get more info $\rightarrow$ Deep Learning Book (Goodfellow et al.)**

| **Aspect** | **ANN** | **CNN** |
|---|---|---|
| Input handling | Flattened vectors | Preserves 2D grid structure |
| Parameters | Fully connected (very large) | Shared weights (efficient) |
| Translation handling | No built-in invariance | Naturally translation-friendly |
| Suitable tasks | Simple, small datasets | Images, speech, spatial data |

## Current Section

## Convolutional Layer: Capturing Local Patterns

Unlike ANNs, which fully connect inputs to neurons, CNNs use the **convolutional layer** to scan small filters (kernels) across the image.

**Definition**

A convolutional layer applies a filter $K \in \mathbb{R}^{m \times n}$ over an input image $I \in \mathbb{R}^{H \times W}$ to produce a feature map $F$:

$$F(i,j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} K(u,v) \cdot I(i+u, j+v)$$

**Output size formula:** For input size $(H, W)$, kernel size $(m, n)$, stride $s$, and padding $p$:

$$H_{out} = \frac{H - m + 2p}{s} + 1, \quad W_{out} = \frac{W - n + 2p}{s} + 1$$

**Stride ($s$):** The number of steps the filter moves when sliding across the image. Larger stride $\rightarrow$ smaller output.
**Padding ($p$):** Adding extra border pixels (usually zeros) around the image so filters can cover edges. Padding preserves input size when needed.

## Convolutional Layer: Capturing Local Patterns

**Filter as a Detective's Magnifying Glass**

Imagine a detective scanning a large painting with a magnifying glass. At each position, the glass only shows a small region, revealing local details (edges, textures). Moving the glass across the painting is like sliding a convolutional filter across an image.

**Numerical Example: 3×3 Filter on 5×5 Image**

- Input: $5 \times 5$, filter: $3 \times 3$, stride = 1, no padding.

$$H_{out} = \frac{5-3}{1} + 1 = 3, \quad W_{out} = \frac{5-3}{1} + 1 = 3$$

So output feature map = $3 \times 3$. Total operations $= 9$ outputs $\times$ 9 multiplications each = 81.

## Convolutional Layer: Capturing Local Patterns

**TensorFlow: Convolution Layer Example**

```python
import tensorflow as tf

# Example: Convolution layer for 28x28 image
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(filters=32,
                           kernel_size=(3,3),
                           activation='relu',
                           input_shape=(28,28,1))
])

model.summary()
# Expected output: Conv2D layer with (3x3x1)x32 + 32 = 320 params
```

**Mistakes**

- Using filters that are too large, losing local focus. - Forgetting to add nonlinearity after convolution. - Misunderstanding filter count: each filter learns one pattern (edge, curve, etc.).

## Convolutional Layer: Capturing Local Patterns

**Caveats**

- Choice of kernel size $(m, n)$, stride, and padding significantly affects output size. - More filters $\rightarrow$ more expressive power, but also more parameters.

**Get more info** $\rightarrow$ **CNN Explainer (Interactive Visualization)**

| **Concept** | **Key Point** |
|---|---|
| Convolution definition | Sliding filter computes local weighted sum |
| Output size formula | $H_{out} = \frac{H-m+2p}{s} + 1$, $W_{out} = \frac{W-n+2p}{s} + 1$ |
| Benefit | Preserves spatial locality, fewer parameters |
| Example | 3×3 filter on 5×5 image $\rightarrow$ 3×3 output |
| Design choices | Kernel size, stride, padding |

## Current Section

## Understanding Kernels in Convolutions

The **kernel** (or filter) is the central building block of CNNs. It is a small matrix of weights that slides across the image to detect patterns such as edges, corners, or textures.

**Definition**

Given an input $I \in \mathbb{R}^{H\times W}$ and a kernel $K \in \mathbb{R}^{m\times n}$, the convolution operation produces a feature map $F$:

$$F(i,j) = \sum_{u=0}^{m-1}\sum_{v=0}^{n-1} K(u,v)\cdot I(i+u,j+v)$$

- $K$ learns to detect a specific feature (e.g., vertical edge). - Multiple kernels $\rightarrow$ multiple feature maps, each capturing different features.

**Kernel as a Detective's Toolkit**

Think of each kernel as a detective's tool designed to spot a certain clue (like fingerprints or footprints). When applied across the entire image, it highlights wherever that clue is found. Different kernels act as different tools for different clues.

## Understanding Kernels in Convolutions

**Numerical Example: 3×3 Kernel on 4×4 Image**

Input $I$:

$$\begin{bmatrix} 1 & 2 & 0 & 1 \\ 3 & 1 & 2 & 2 \\ 0 & 1 & 3 & 1 \\ 2 & 0 & 1 & 2 \end{bmatrix}$$

Kernel $K$:

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

First convolution output at $(0,0)$:

$$(1\times 1)+(2\times 0)+(0\times -1)+(3\times 1)+(1\times 0)+(2\times -1)+(0\times 1)+(1\times 0)+(3\times -1)$$

$$=1+0+0+3+0-2+0+0-3$$

$$=2$$

This kernel acts as a **vertical edge detector**, producing strong responses where vertical edges exist.

## Understanding Kernels in Convolutions

**Mistakes**

- Treating kernels as arbitrary numbers: in CNNs they are **learned**, not manually chosen.
- Confusing convolution with element-wise multiplication – convolution involves sliding and summing.

**Caveat**

Each kernel specializes in detecting one feature. A CNN needs multiple kernels in the same layer to capture a rich set of features.

**Get more info → Dive into Deep Learning: Convolution Layer**

| **Concept** | **Key Point** |
|---|---|
| Kernel definition | Small matrix of weights scanning the image |
| Kernel role | Detects local features (edges, corners, textures) |
| Example | 3×3 edge-detection kernel applied to 4×4 image |
| Note | Multiple kernels → multiple feature maps |

## Current Section

## Stride and Padding Explained

After understanding the kernel, two key hyperparameters control how kernels slide and how outputs are shaped: **stride** and **padding**.

### Definitions

- **Stride (*s*)**: Number of pixels the filter moves each step when scanning across the image. Larger stride $\rightarrow$ smaller output, more downsampling.
- **Padding (*p*)**: Extra border pixels (often zeros) added around the input image. Padding helps kernels cover the image edges and can preserve spatial dimensions.

**Output size formula:**

$$H_{out} = \frac{H - m + 2p}{s} + 1, \quad W_{out} = \frac{W - n + 2p}{s} + 1$$

where $H \times W$ = input size, $m \times n$ = kernel size.

### Stride and Padding as Walking on Tiles

Stride is like the size of each step you take while walking on floor tiles: small steps cover more tiles, big steps skip some. Padding is like adding extra tiles around the border so you can also step at the very edges.

## Stride and Padding Explained

### Numerical Examples

Input: $5 \times 5$ image, Kernel: $3 \times 3$
- **Stride = 1, Padding = 0** (Output: $3 \times 3$ ):

$$H_{out} = \frac{5-3}{1} + 1 = 3, \quad W_{out} = 3$$

- **Stride = 2, Padding = 0** (Output: $2 \times 2$ ):

$$H_{out} = \frac{5-3}{2} + 1 = 2, \quad W_{out} = 2$$

- **Stride = 1, Padding = 1** (Output: $5 \times 5$, same as input size):

$$H_{out} = \frac{5-3+2}{1} + 1 = 5, \quad W_{out} = 5$$

# Stride and Padding Explained

**Mistakes**

- Assuming stride $> 1$ always preserves useful features. Large strides may skip critical details. - Using excessive padding can introduce artificial borders in the feature maps.

**Caveat**

The right combination of stride and padding balances efficiency and accuracy. Small stride with padding often works best for image recognition tasks.

**Get more info** $\rightarrow$ **CS231n: Stride and Padding Explanation**

| **Concept** | **Key Point** |
|---|---|
| Stride | Step size of kernel movement |
| Padding | Extra border pixels added to input |
| Output size formula | $H_{out} = \frac{H-m+2p}{s} + 1,\ W_{out} = \frac{W-n+2p}{s} + 1$ |
| Example | $5 \times 5$ input, $3 \times 3$ kernel with different stride/padding |

## Current Section

## Types of Padding in TensorFlow

When designing CNNs in TensorFlow/Keras, the choice of **padding** directly impacts output size and feature coverage. The central question: **How can we control whether edges of the image are included in the convolution?**

### Types of Padding

TensorFlow supports three main padding schemes:

- **Valid Padding** ("valid") No padding added. Output shrinks depending on kernel size.

$$H_{out} = \frac{H-m}{s} + 1, \quad W_{out} = \frac{W-n}{s} + 1$$

- **Same Padding** ("same") Pads input so output size = input size (when stride = 1). Ensures every pixel contributes equally.

$$H_{out} = \left\lceil \frac{H}{s} \right\rceil, \quad W_{out} = \left\lceil \frac{W}{s} \right\rceil$$

- **Causal Padding** ("causal") Used in 1D convolutions (e.g. time series). Ensures output at time $t$ depends only on current and past inputs, not the future.

$$H_{out} = \left\lceil \frac{H}{s} \right\rceil, y[t] = \sum_{k=0}^{K-1} w[k] \cdot x[t-k], p_{left} = K-1, p_{right} = 0$$

## Types of Padding in TensorFlow

**VALID Padding Example (7×7 Input, 3×3 Kernel)**

Parameters: $H = W = 7$, $m = n = 3$, padding $p = 0$
**Case 1: Stride = 1**

$$H_{out} = \frac{7-3+2\cdot 0}{1} + 1 = 5, \quad W_{out} = 5$$

Output size = $5 \times 5$
**Case 2: Stride = 2**

$$H_{out} = \frac{7-3+2\cdot 0}{2} + 1 = \frac{4}{2} + 1 = 3, \quad W_{out} = 3$$

Output size = $3 \times 3$

# Types of Padding in TensorFlow

**SAME Padding Example (7×7 Input, 3×3 Kernel)**

Output formula:

$$H_{out} = \frac{H - m + (p_{top} + p_{bottom})}{s} + 1, \; W_{out} = \frac{W - n + (p_{left} + p_{right})}{s} + 1$$

TensorFlow rule:

$$H_{out} = \left\lceil \frac{H}{s} \right\rceil, \; W_{out} = \left\lceil \frac{W}{s} \right\rceil$$

Total padding:

$$p_H = (H_{out} - 1) \cdot s + m - H, \; p_W = (W_{out} - 1) \cdot s + n - W$$

Symmetric split:

$$p_{top} = \lfloor p_H/2 \rfloor, \; p_{bottom} = \lceil p_H/2 \rceil$$

$$p_{left} = \lfloor p_W/2 \rfloor, \; p_{right} = \lceil p_W/2 \rceil$$

Asymmetric option:

$$p_{top} + p_{bottom} = p_H, \; p_{left} + p_{right} = p_W$$

Example (7,3, stride=2): $H_{out} = \lceil 7/2 \rceil = 4$, $p_H = 2$. Choices: $(1,1)$ or $(0,2)$.

## Types of Padding in TensorFlow

**Padding as Window Frames**

- **Valid:** Like looking through a window without any frame extension. You only see the inner portion, edges are cut off.
- **Same:** Like adding a frame extension around the window so you can still see the full outside view without losing the borders.
- **Causal:** Like drawing window blinds that only open toward the past direction. You can only see current and earlier scenery, never what comes in the future.

## Types of Padding in TensorFlow

**TensorFlow Example: Valid vs Same Padding**

```python
import tensorflow as tf

# Valid padding
model_valid = tf.keras.Sequential([
    tf.keras.layers.Conv2D(1, (3,3), padding='valid',
                           input_shape=(5,5,1))
])

# Same padding
model_same = tf.keras.Sequential([
    tf.keras.layers.Conv2D(1, (3,3), padding='same',
                           input_shape=(5,5,1))
])

print("Valid output:", model_valid.output_shape)
print("Same output:", model_same.output_shape)
# Expected: (None, 3, 3, 1) vs (None, 5, 5, 1)
```

## Types of Padding in TensorFlow

**Common Mistakes**

- Believing "same" always keeps size identical even when stride $> 1 \rightarrow$ Correct: "same" keeps proportional size using ceiling division, exact equality only holds when stride = 1.
- Using "valid" when edge details are important $\rightarrow$ Correct: use "same" padding to preserve border information.
- Applying "causal" padding in 2D/3D convolutions $\rightarrow$ Correct: "causal" is only valid for 1D temporal convolutions.

**Caveats**

- Choice of padding affects both computational cost and accuracy.
- "Same" padding can introduce artificial borders if kernel is large.
- "Causal" ensures temporal correctness but may reduce efficiency.

**Get more info $\rightarrow$ Understanding masking & padding**

## Summary: Types of Padding

| **Padding Type** | **Key Explanation** |
|---|---|
| Valid | No padding. Output shrinks depending on kernel. |
| Same | Pads input so output $\approx$ input size (stride=1 exact). |
| Causal | For 1D only. Ensures future inputs never affect past outputs. |
| Mistake | Assuming "same" = identical size for all strides. |