Unlocking Image Secrets with Convolutional Neural Networks

Minor In AI, IIT Ropar
23rd May, 2025

Welcome back, aspiring AI enthusiasts! In our previous sessions, we got a feel for how neural networks, specifically Multi-Layer Perceptrons (MLPs), can tackle classification tasks. We saw how they can take input, perform calculations through layers of 'neurons,' and produce an output, like classifying handwritten digits.

But we also hit a snag when thinking about using MLPs directly for images. Remember how we talked about potentially flattening an image – turning that grid of pixels into a long, single line of numbers – before feeding it into an MLP? While technically possible, this approach throws away crucial information. An image isn't just a random collection of pixels; it's a grid where the *location* of a pixel relative to its neighbors matters. Flattening an image means losing this 'spatial' information. Where is the nose on the face? Where is the top edge of the digit '7'? Flattening makes these relative positions disappear, making it much harder for the network to learn meaningful patterns.

So, we need a different approach, one that respects the 2D grid structure of images right from the start. Enter the world of Convolutional Neural Networks, or CNNs.

Seeing Images Differently: The Convolution Idea

Imagine trying to figure out what a picture is, not by looking at the whole thing at once, but by carefully examining small pieces of it, one by one, and noting down what you see in each piece. This is the core idea behind a key operation in CNNs called **Convolution**.

Let's look at a visual example:

See that small square box moving over the larger grid? The larger grid represents our input image (or at least a part of it). The small moving box is what we call a **filter** or **kernel**. The operation involves the filter stopping at different positions on the image and performing a specific calculation with the pixels it currently covers. The picture shows how the filter moves across, first horizontally, then shifting down and moving horizontally again.

This process doesn't produce a single output number like an MLP layer does for a single input vector. Instead, for each position the filter stops, it produces *one* number in a new grid – the output. Notice how the input is a grid (a matrix), and the output is also a grid (another matrix)! This is the fundamental building block of CNNs. But how does this calculation actually work? And what does it tell us?

Dissecting the Convolution Operation

Let's break down what's involved in this operation using the example we explored:

Look at the input image side. Notice the extra layer of white around the original image data? This is called **Padding**. It's like adding bubble wrap around the image – adding extra pixels, usually zeros, around the border.

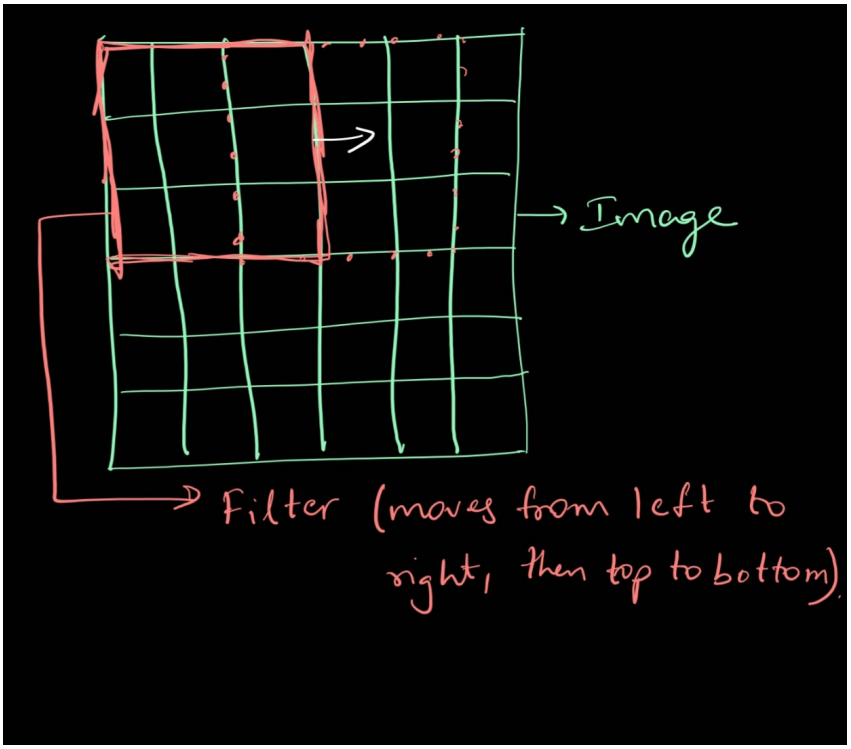

Figure 1: An image showing a small square filter moving across a larger image grid from left to right, top to bottom, with a step size. This illustrates the fundamental convolution operation.

- **Padding (P):** The number of pixel layers added around the border. In our example, we added one layer of zeros all around, so the padding (P) is 1. Padding is often used to help the filter process pixels near the edges of the image and can help control the output size. We briefly saw other padding types (like repeating edge values), but **zero padding** is the most common.

For a standard color image, we don't just have one grid of numbers representing brightness. We typically have three grids: one for Red, one for Green, and one for Blue intensities.

- **Input Channels (D_I):** The number of 'layers' or grids in the input image. For a color RGB image, $D_I = 3$. For a grayscale image like MNIST digits, $D_I = 1$. The input size is often described as Height (H_I) $\times$ Width (W_I) $\times$ Depth (D_I). In the initial example, the *original* image was 5×5, but with padding of 1, the *effective* input processed by the filter was 7×7 (including the padding) across 3 channels.

Next, consider the filters. We saw not just one, but often multiple filters used simultaneously.

- **Number of Filters (K):** How many different filters we apply to the image. In the animation, there were two filters. Each filter is designed to look for something potentially different in the image.
- **Filter Size (F):** The dimensions of the square filter (e.g., 3×3, 5×5). In our example, the filter size was 3×3. Importantly, a filter also has depth. This depth must match the number of input channels.
- **Filter Components:** Each filter is actually composed of multiple 2D matrices – one for each input channel. If the input has 3 channels (RGB), a filter of size 3×3 will have three 3×3 components, one dedicated to processing the Red channel's part of the image, one for Green, and one for Blue.

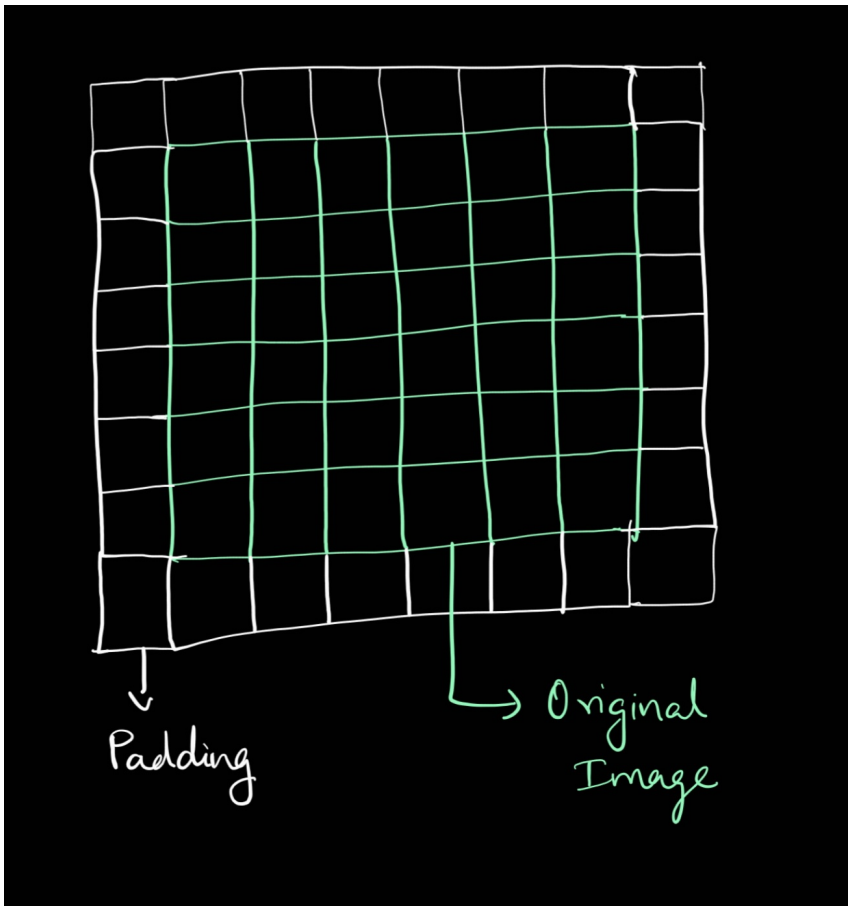

Figure 2: An image showing the input image with padding. This helps visualize the key components involved in a convolution layer.

Finally, let's revisit the movement of the filter.

- **Stride (S):** How many pixels the filter shifts each time it moves. In the animation, the filter jumped two pixels at a time horizontally, and then two pixels down vertically. So, the stride (S) was 2. A stride of 1 means the filter moves one pixel at a time, resulting in significant overlap between successive positions. A larger stride means larger jumps and less overlap.

Okay, we have the pieces: Input Image (with H_I, W_I, D_I, and optional P), Filters (with K, F, and components matching D_I), and Stride (S). How does the calculation happen?

When a filter lands on a spot in the image, the operation is **element-wise multiplication and summation**. For each input channel (R, G, B), you take the 3×3 section of the image currently under the filter's component for that channel. You multiply the numbers in the image section by the corresponding numbers in the filter component, element by element.

- Example (R channel part): $(\text{Image}[0,0] \times \text{Filter}[0,0]) + (\text{Image}[0,1] \times \text{Filter}[0,1]) + \cdots + (\text{Image}[2,2] \times \text{Filter}[2,2]) = \text{Sum}_R$
- Do this for the G channel: Sum_G
- Do this for the B channel: Sum_B

Then, you **sum up the results from all channels**: $\text{Total\_Sum} = \text{Sum}_R + \text{Sum}_G + \text{Sum}_B$.

Finally, each filter typically has a single **bias** value associated with it.

- **Bias:** A single number added to the Total_Sum.

The final output value for that filter at that position is: $\text{Output\_Value} = \text{Total\_Sum} + \text{Bias}$.

This output value goes into the corresponding position in the output grid for that specific filter. Repeat this process for every position the filter stops at according to the stride. Then, repeat the entire process for the *next* filter to get its corresponding output grid.

In our lecture example, for the first filter's first position, the calculations based on the animation values (after clarification during the lecture) were:

- Sum from R channel processing: -3
- Sum from G channel processing: -1
- Sum from B channel processing: -2

Total sum from all channels $= (-3) + (-1) + (-2) = -6$. The bias for this filter was 0. So, the final output value = Total Sum + Bias $= -6 + 0 = -6$. This matched the first output value shown.

For the second position calculated (yielding 10): The sums from the three channel blocks were 6, 2, and 1 respectively. The bias was 1. Total sum from all channels $= 6 + 2 + 1 = 9$. Final output value = Total Sum + Bias $= 9 + 1 = 10$. This matched the second output value.

This element-wise multiply and sum across the filter area and all channels, plus bias, is the **convolution operation**.

Convolution as Feature Detection: Why Does This Matter?

Okay, so we do this mathematical operation. What does the resulting output grid of numbers represent? This is the intuitive part!

Remember how recognizing a handwritten digit '7' involved spotting a horizontal line and a slanted line? These lines are *features*. Convolution helps us detect such features.

Think of the numbers inside a filter as a *pattern* or a *template*. When you perform the element-wise multiplication and sum, you are essentially checking how well the section of the image currently under the filter *matches* the filter's pattern.

- If the image section strongly matches the filter's pattern (e.g., a filter designed to detect horizontal lines aligns perfectly with a horizontal line in the image), the resulting sum will be a large positive number.
- If it strongly matches the *opposite* pattern (e.g., a horizontal line filter on a vertical line), the sum might be a large negative number.
- If there's no strong match (e.g., the filter is over a blank area or a different shape), the sum will be close to zero.

The numbers in the output grid for a specific filter tell you **how strongly the feature that filter is looking for is present in each corresponding region of the original image**. A high positive value means "Yes, that feature is strongly here!" A value near zero means "No, that feature isn't really here."

Since we use multiple filters (K filters), the first convolutional layer produces K output grids, often called **feature maps**. Each feature map highlights where a *specific* feature (corresponding to one filter) is located and how intensely it's present across the image.

Stacking Convolutional Layers

These feature maps generated by the first convolutional layer aren't our final answer. They represent simple features like edges or corners. Just like how we build complex shapes from simple lines, a CNN builds up the ability to recognize complex patterns by stacking multiple convolutional layers.

The output feature maps from the first layer become the *input* to the second convolutional layer. The filters in the second layer will now operate on these 'feature maps' instead of raw pixel values. A filter in the second layer might learn to detect patterns *of edges* from the first layer's outputs – for example, combining edges to recognize a corner or a curve.

Each subsequent convolutional layer processes the feature maps from the previous layer, learning to detect more complex and abstract features. The number of channels (depth) increases after each convolutional layer, matching the number of filters (K) used in that layer.

We also typically apply an **activation function** (like ReLU, which sets any negative values to zero) after each convolution operation to introduce non-linearity, just like in MLPs.

Designing Your CNN: Choices and Constraints

When building a CNN, you have several design choices to make for each convolutional layer:

1. **How many Convolutional Layers?** More layers can learn more complex features, but also make the network bigger and harder to train. There's no fixed rule; it's often experimental, starting with a few and adding more if needed (similar to deciding the number of layers in an MLP).
2. **How many Filters (K) in each layer?** More filters allow the layer to detect more features. Again, this is often determined experimentally. You typically start with a

reasonable number (e.g., 32 or 64) and increase if necessary.

3. **What Filter Size (F)?** Smaller filters (like 3×3 or 5×5) are common, especially for smaller images. Larger filters ($7 \times 7, 11 \times 11$) might be used for very large input images to capture larger patterns initially. Heuristics suggest using smaller filters when the input image size is small, and larger filters for larger inputs.
4. **What Stride (S) and Padding (P)?** These affect how the filter moves and the size of the output feature maps.
5. **What Activation Function?** ReLU is a popular default choice after convolutional operations.

Choosing these values isn't always straightforward. As we saw with the calculation example:

Given Input Size (H_I, W_I, D_I) and parameters (K, F, S, P), the Output Size (H_O, W_O, D_O) is calculated as:

- $D_O = K$ (Number of Filters)
- $H_O = \frac{H_I + 2P - F}{S} + 1$
- $W_O = \frac{W_I + 2P - F}{S} + 1$

Important Note: The results for H_O and W_O must be whole numbers (integers), and they must be positive. You can't have a fractional or negative size for your output grid! This constraint limits the valid combinations of F, P, and S for a given input size.

We saw an example where given an Input size of $28 \times 28 \times 3$ and a desired Output size of $24 \times 24 \times 16$, we needed to find feasible values for K, F, P, and S.

- From the output depth, we know K must be 16 ($D_O = K$).
- Using the formula for H_O (and W_O, since they are the same size): $\frac{28+2P-F}{S} + 1 = 24$. This simplifies to $\frac{28+2P-F}{S} = 23$, or $28 + 2P - F = 23S$.
- We have one equation and three unknowns (F, P, S). It's impossible to find unique solutions. So, we make common assumptions:
  - Assume $S = 1$: $28 + 2P - F = 23$. Rearranging: $F = 5 + 2P$. If we also assume $P = 0$, then $F = 5$. If $P = 1$, then $F = 7$. Both $(F = 5, P = 0, S = 1)$ and $(F = 7, P = 1, S = 1)$ are potentially valid sets of parameters for this size transformation.
  - Assume $S = 2$: $28 + 2P - F = 23 \times 2 = 46$. Rearranging: $F = 2P - 18$. If $P = 0$, $F = -18$ (not possible, filter size must be positive). If $P = 1$, $F = -16$ (not possible). Larger P might yield positive F, but F is usually kept relatively small. This shows not all assumptions lead to valid filter sizes.

These calculations show that the design choices aren't completely arbitrary; they must satisfy the output size calculation formula and result in valid dimensions.

From Features Back to Classification: The Classification Head

After passing the image through several convolutional layers, we end up with a set of feature maps. This output block (e.g., $3 \times 3 \times 3$ in one of our examples, or $24 \times 24 \times 16$ in the calculation example) is rich in information about the features present in the original image.

But our goal is classification! We need a final output vector of probabilities (one probability for each possible class, summing to 1), not a 3D block of feature intensities. How do we go from

the output block of the last convolutional layer to a classification vector?

Remember the problem we started with? Flattening an image loses spatial information. But after several convolutional layers, we are no longer dealing with raw pixel data. We are working with *feature maps* – grids indicating the intensity of abstract features at different locations. At this stage, the exact spatial arrangement of these feature intensities is less critical than the *combination* of features present across the entire set of maps.

So, the solution is simple: we **flatten** the output block of the last convolutional layer! This flattened vector contains all the feature intensities extracted by the convolutional layers. It effectively summarizes the presence and strength of various learned features across the original image.

This flattened vector is then fed into a standard **MLP**. This MLP acts as the **classification head** of the CNN. It takes the feature vector as input and maps it to the desired number of output classes. For a multi-class classification problem with, say, 4 classes, the MLP would have an input layer size equal to the size of the flattened vector (e.g., 27) and an output layer size equal to the number of classes (e.g., 4). This final output vector represents raw scores for each class. Passing it through a **Softmax** function converts these scores into a probability distribution.

The Complete CNN Architecture for Classification

So, a typical CNN architecture for image classification looks like this:

1. **Feature Extractor:** A stack of one or more Convolutional Layers (each usually followed by an activation function like ReLU). These layers extract increasingly complex features from the image.
2. **Classification Head:** An MLP, starting with a **Flatten** layer to convert the final feature block into a vector, followed by one or more **Dense** layers (the standard MLP layers), ending with an output Dense layer matching the number of classes and typically a Softmax activation for classification.

The key distinction is that the initial layers (Feature Extractor) specifically process the image's spatial structure using convolution, while the final layers (Classification Head) are the standard MLP we've seen before, operating on the extracted features to make the final class prediction.

And how do the numbers inside all those filters (the weights) get determined? Just like the weights in an MLP, they are **learnable parameters** that are adjusted during training using **backpropagation** and optimization algorithms like gradient descent, based on how well the network is performing the classification task. The network *learns* which features are important for the task and what patterns to look for by adjusting the filter values.

This gives us a complete picture of how a Convolutional Neural Network can take an image input, extract meaningful features while preserving some sense of spatial relationships initially, and then use those features to perform classification.

In the next session, we'll dive deeper into other components commonly found in CNNs and look at some famous CNN architectures that have achieved incredible results in image recognition tasks.

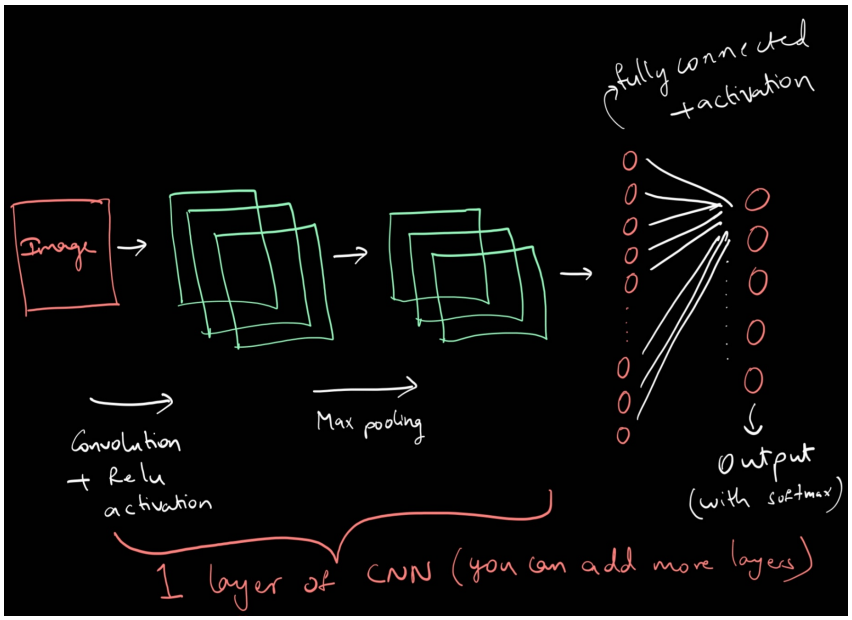

Figure 3: A high-level diagram showing the complete flow: Input Image → [Conv Layers + ReLU] x N (Feature Extractor) → Flatten → [Dense Layers + ReLU] x M (Classification Head) → Output Layer (Softmax) → Classification Probabilities.