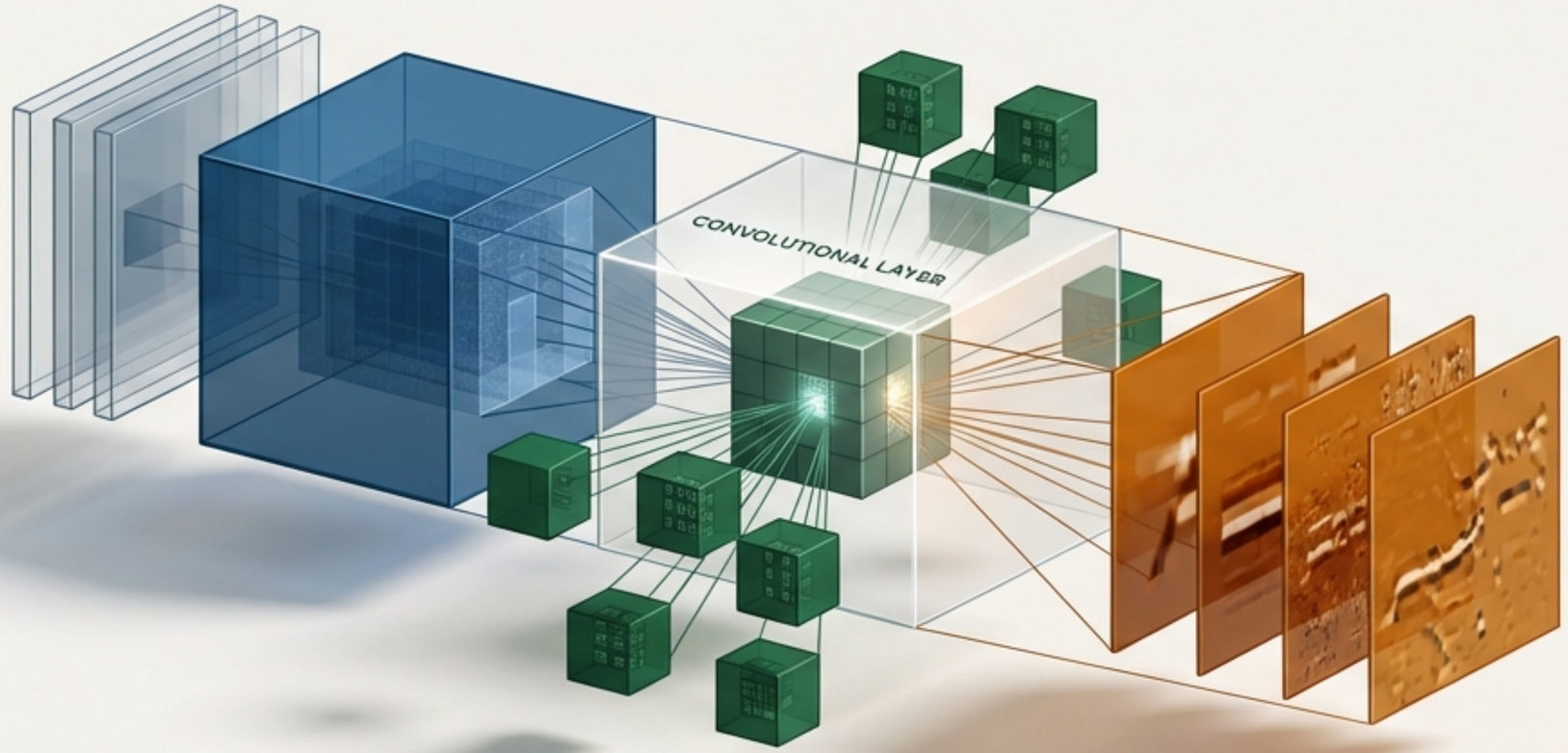
Building a Convolutional Layer, Line by Line

A step-by-step guide to the mathematics and Python code behind one of deep learning's fundamental building blocks.

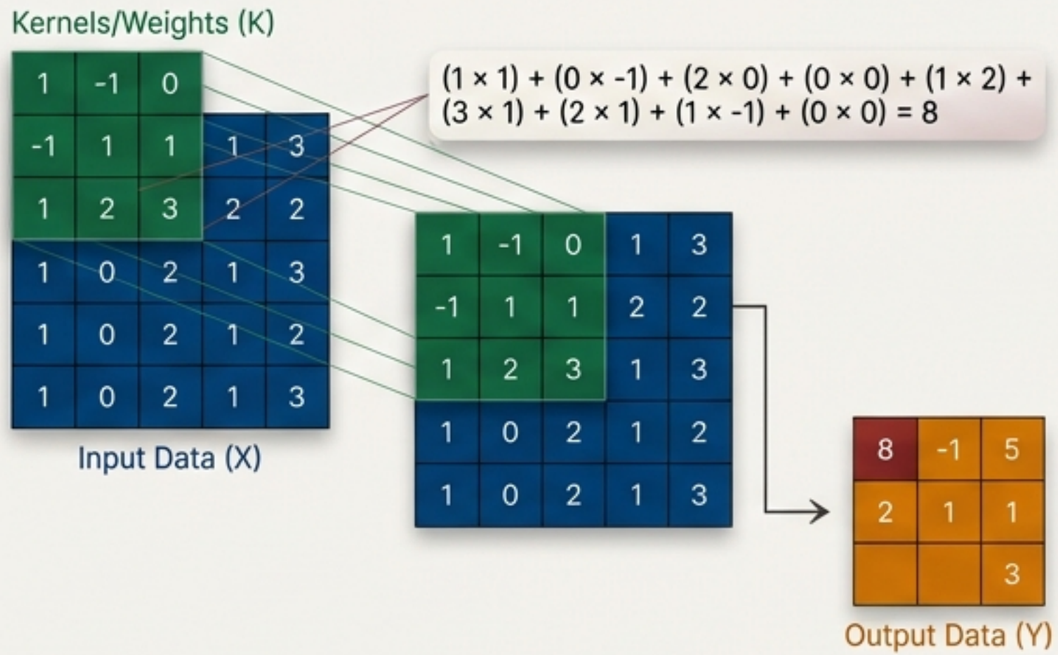


This presentation demystifies the CNN layer. We will move past the abstractions of popular libraries and build our own, starting from first principles. By the end, you will understand not just *what* it does, but precisely *how* it works.

Let’s Be Precise: Convolution vs. Cross-Correlation

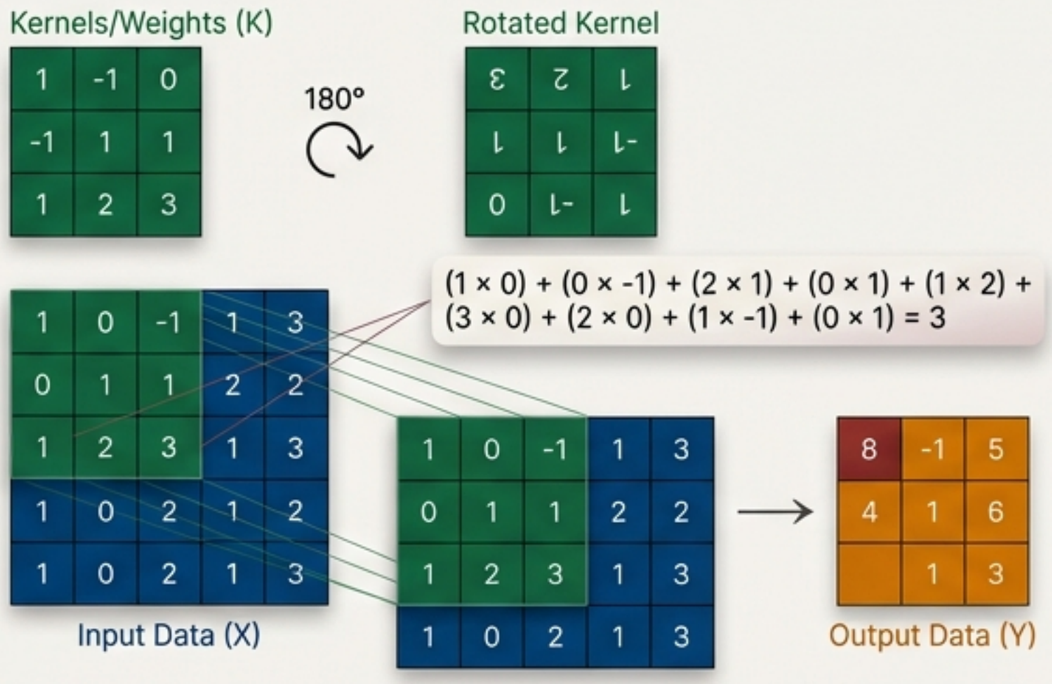
What is commonly called ‘convolution’ in the context of neural networks is, mathematically, **cross-correlation**. The distinction is crucial for understanding the underlying operations and implementing them correctly.

☆ Cross-Correlation



The kernel slides over the input, performing element-wise multiplication and summation. This is the operation we will primarily use.

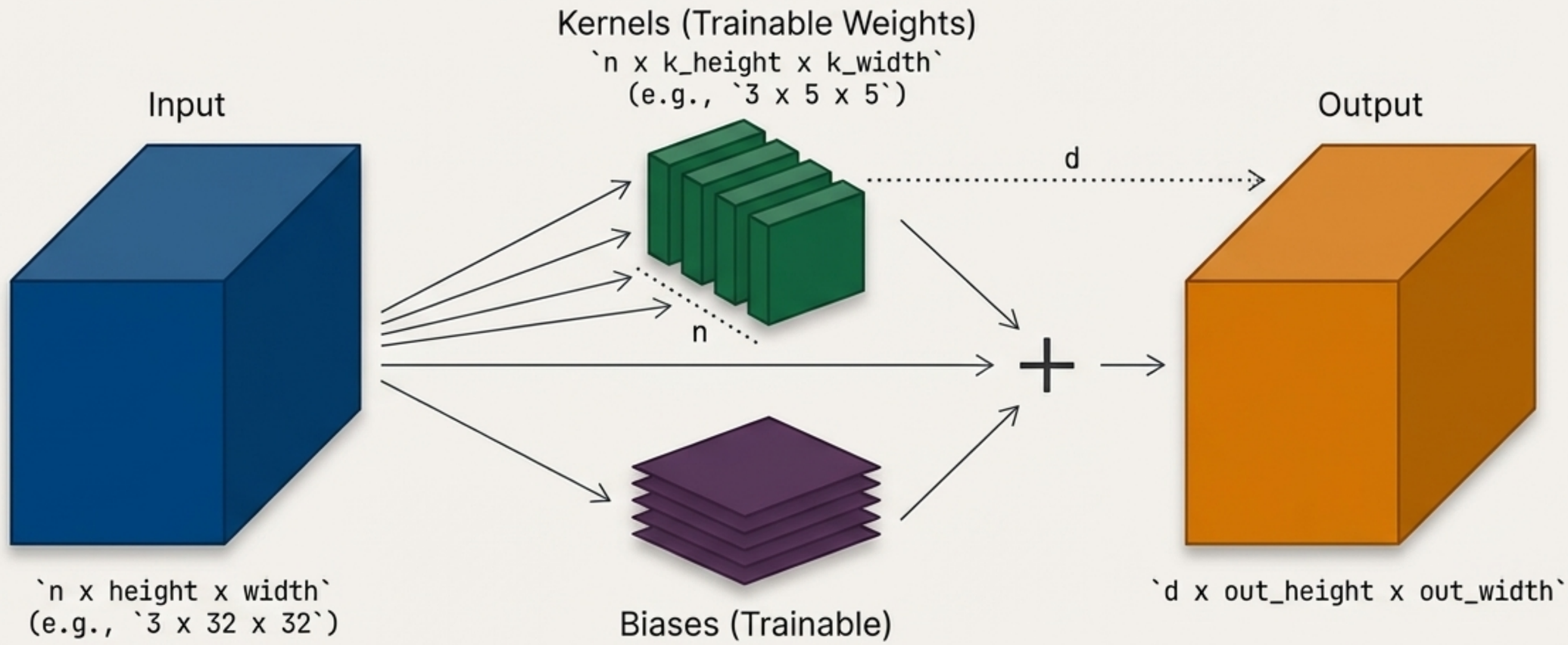
∗ Convolution



True convolution involves rotating the kernel by 180° first. We will see this operation make a surprising appearance during backpropagation.

The Anatomy of a Convolutional Layer

A convolutional layer processes data in 3D volumes. Its goal is to learn a set of filters (kernels) that detect specific features in the input.

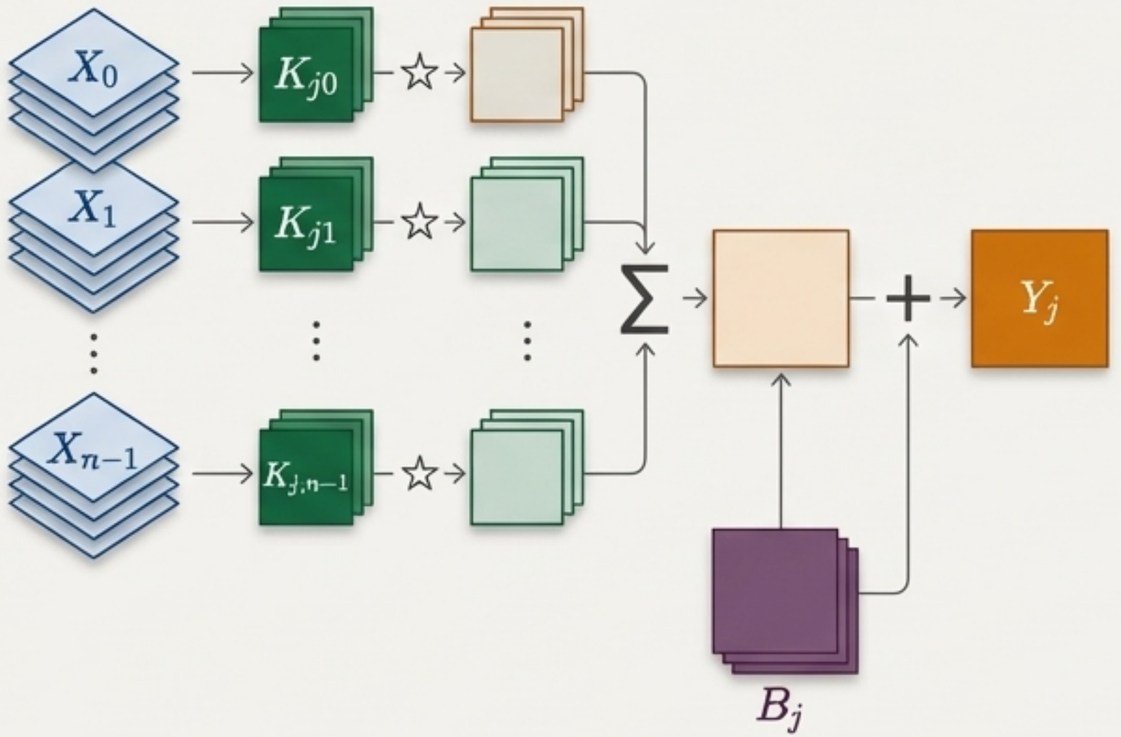


The Forward Pass: From Input to Output Map

Each slice of the output volume is generated by sliding one kernel across the entire input volume, performing cross-correlation at each depth, summing the results, and adding a bias.

$$Y_j = B_j + \sum_{i=0}^{n-1} (X_i \star K_{ji})$$

- Y_j: The j-th output feature map.
- B_j: The bias matrix associated with the j-th kernel.
- X_i: The i-th channel of the input volume.
- K_{ji}: The i-th channel of the j-th kernel.
- $\star$: The cross-correlation operator.
- $\sum$: Summation across all input channels (i).

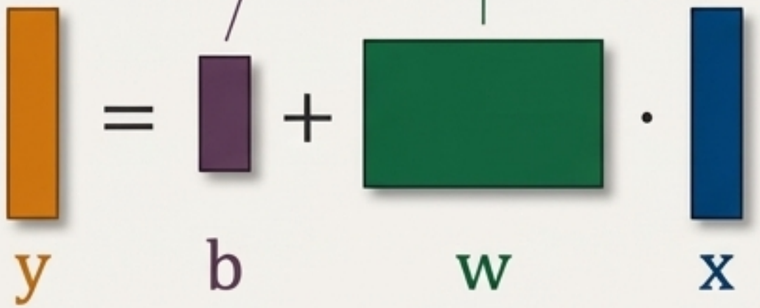


A Deeper Connection: The CNN as a Generalised Dense Layer

The forward propagation equation $Y = B + X \star K$ looks remarkably familiar.
It's a generalised form of the dense layer equation $y = b + w \cdot x$.

Dense Layer

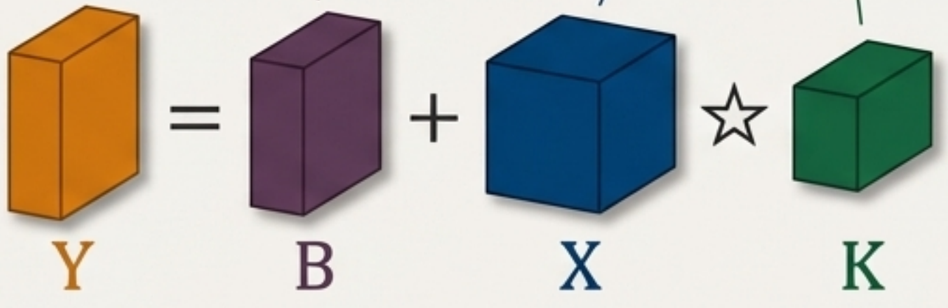
$$y = b + w \cdot x$$



$\approx$

Convolutional Layer

$$Y = B + \sum(X \star K)$$



Think about it: if each 'matrix' in our equation was just a single number (a 1x1 matrix), the cross-correlation operator ('☆') would simply become multiplication ('*'), and the 'matrix dot product' would become a regular vector dot product ('·').

A dense layer is just a special case of a convolutional layer where the kernel size is 1x1 and operates on flattened inputs. The convolutional layer generalises this concept to spatial data.

Implementing the Forward Pass in Python

```
# ConvolutionalLayer.forward(self, input_data)

self.input = input_data
# Initialise output with biases
self.output = np.copy(self.biases)

# Iterate over each kernel (output depth)
for i in range(self.depth):
    # Iterate over each input channel
    for j in range(self.input_depth):
        self.output[i] += correlate2d(
            self.input[j], self.kernels[i, j], mode='valid'
    )
return self.output
return self.output
```

1. Starts with Bias

We start with the bias term for each output map.

2. Mirrors the Math

The loops directly mirror the `i` and `j` indices from the mathematical summation $\sum_{i=0}^{n-1}$.

3. Cross-Correlation in Action

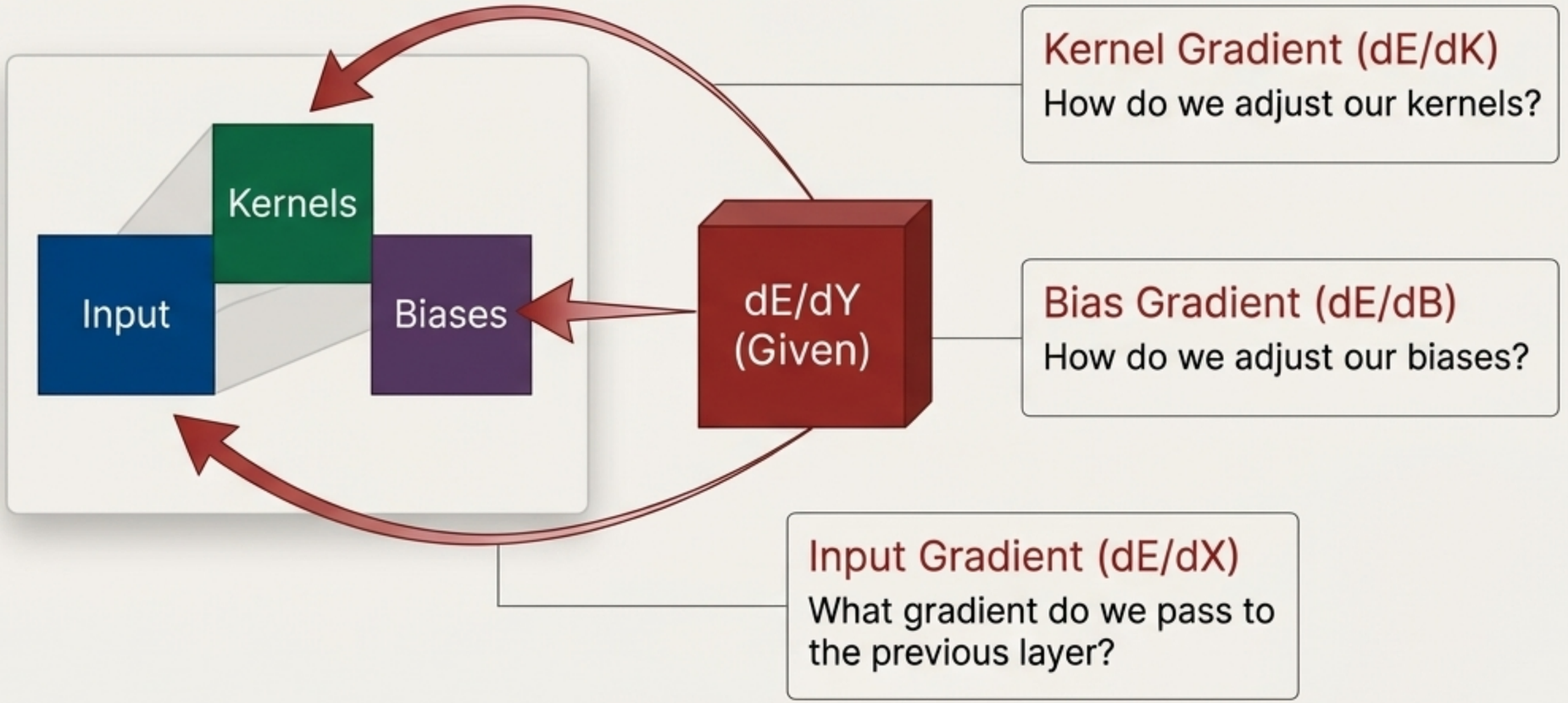
We use SciPy's function to perform the cross-correlation (☆).

4. Valid Padding

This ensures the kernel slides only where it fully overlaps the input, producing an output of size `input_size - kernel_size + 1`.

The Backward Pass: How the Layer Learns

To update our trainable parameters (kernels and biases), we need to calculate how a change in each parameter affects the total network error (**E**). This is done by computing gradients. We are given the gradient of the error with respect to our layer's output (**dE/dY**) and must compute three things:



Deriving the Kernel Gradient (A Simplified Case)

The Setup

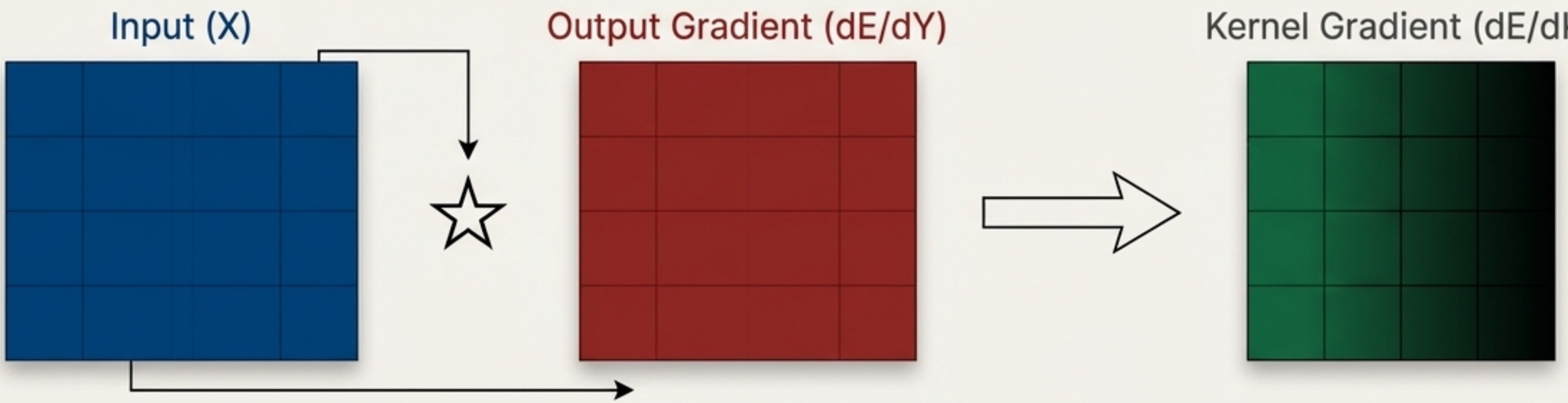
Let's simplify the forward pass to a single input and single kernel: Y = B + X ☆ K. Our goal is to find dE/dK, given dE/dY.

The Logic

By applying the chain rule to each element of the kernel (K_ij), we find that the gradient dE/dK_ij is a sum of terms involving elements from X and dE/dY.

The Insight

When we assemble these individual gradient terms, a familiar pattern emerges.



$$dE/dK = X \star dE/dY$$

The gradient of the kernel is the cross-correlation between the layer's input (X) and the gradient of the error with respect to the layer's output (dE/dY).

The Complete Backpropagation Formulae

Using the insight from our simplified case, we can now state the gradients for the full multi-channel layer.

Kernel Gradient

The gradient for a specific kernel channel K_{ji} is the cross-correlation of the corresponding input channel X_i with the output gradient dE/dY_j.

$$\frac{\mathrm{d}E}{\mathrm{d}K_{ji}} = X_i \star \frac{\mathrm{d}E}{\mathrm{d}Y_j}$$

Bias Gradient

This is the simplest. The gradient for each bias B_j is simply the gradient of its corresponding output map dE/dY_j.

$$\frac{\mathrm{d}E}{\mathrm{d}B_j} = \frac{\mathrm{d}E}{\mathrm{d}Y_j}$$

Input Gradient

This is the most surprising result. The gradient passed back to the previous layer's input channel X_j is the sum of the output gradients dE/dY_i *convolved* with the kernels K_{ij}.

$$\frac{\mathrm{d}E}{\mathrm{d}X_j} = \sum_{i=0}^{d-1} \left(\frac{\mathrm{d}E}{\mathrm{d}Y_i} * K_{ij} \right)$$

Key Takeaway: Notice the elegant symmetry: the forward pass uses cross-correlation (☆), while the input gradient calculation in the backward pass uses true convolution (∗).

Implementing the Backward Pass in Python

```python
# ConvolutionalLayer.backward(self, output_gradient, learning_rate)

kernels_gradient = np.zeros(self.kernels.shape)
input_gradient = np.zeros(self.input.shape)

# Loop for dE/dK and dE/dX
for i in range(self.depth):
    for j in range(self.input_depth):
        # Calculate kernel gradient
        kernels_gradient[i, j] = correlate2d(self.input[j], output_gradient[i], mode='valid')
        # Calculate input gradient
        input_gradient[j] += convolve2d(output_gradient[i], self.kernels[i, j], mode='full')

# Update parameters
self.kernels -= learning_rate * kernels_gradient
self.biases -= learning_rate * output_gradient  # Since dE/dB = dE/dY

return input_gradient
```

1 **Kernel Gradient via Correlation:** Implements `dE/dK = X ☆ dE/dY`.

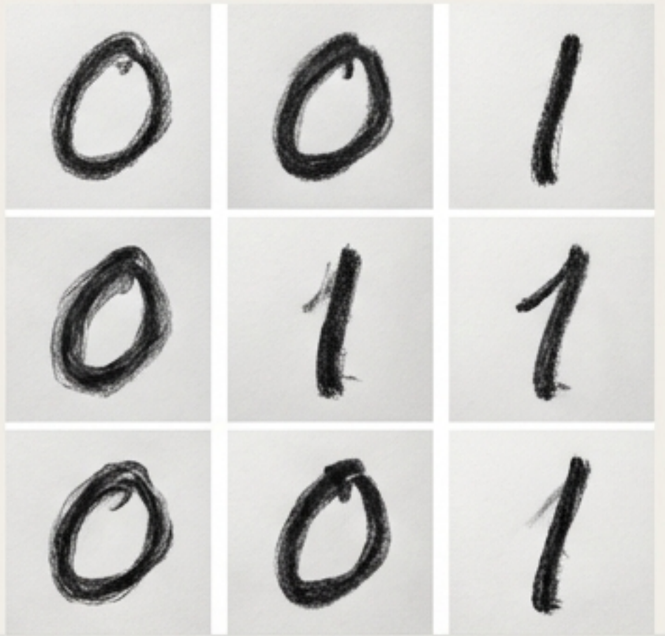
2 **Input Gradient via Convolution:** Implements `dE/dX = dE/dY * K`. Note the use of `mode='full' to match the mathematics of convolution for gradients.

3 **Gradient Descent Update:** The final lines show the simple update rule for kernels and biases.

Putting It All Together: Solving the MNIST Challenge

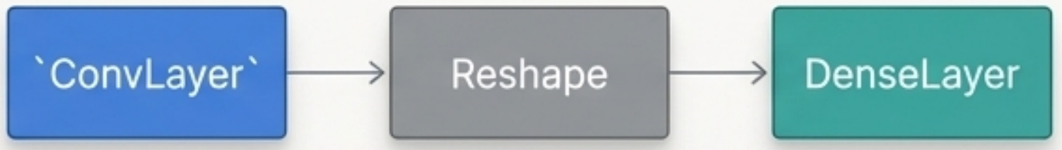
The Goal

We will use our custom convolutional layer to build a network that can classify handwritten digits from the MNIST dataset. For simplicity, we will focus on a binary classification task: distinguishing "0" from "1".



The Missing Pieces

- Our `ConvLayer` is the star, but a full network needs a supporting cast:
- ""**Reshape Layer**"": To flatten the 3D output of our convolutional layer into a 1D vector for a dense layer."
- ""**Sigmoid Activation**"": To introduce non-linearity and bound outputs between 0 and 1.
- ""**Binary Cross-Entropy Loss**"": A suitable loss function for a binary classification problem.



The Supporting Cast: Essential Network Components

Reshape Layer

Flattens a 3D volume to a 1D vector (forward pass) and reverses the operation (backward pass).

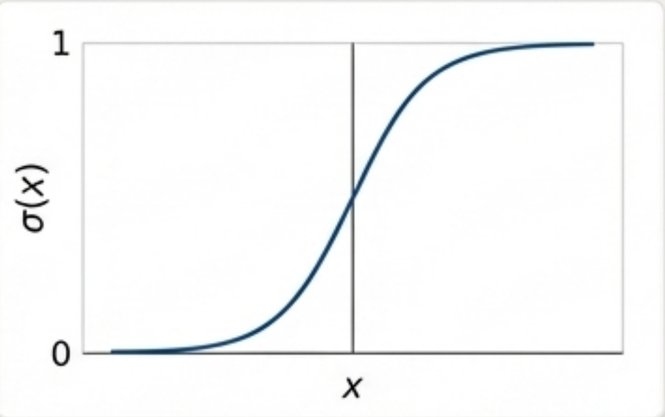
```
# forward
return np.reshape(input,
self.output_shape)

# backward
return np.reshape(output_gradient,
self.input_shape)
```

Sigmoid Activation

Squashes values to a (0, 1) range.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x) * (1 - \sigma(x))$$



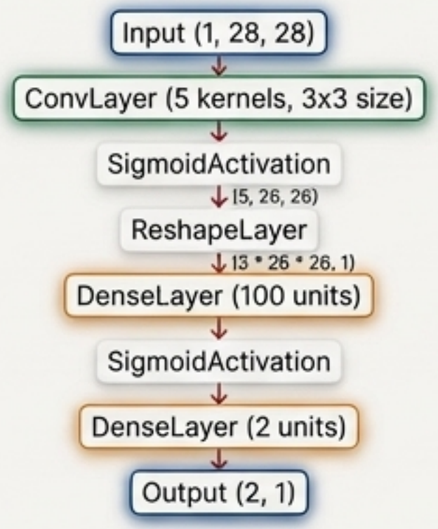
Binary Cross-Entropy Loss

Measures the error for a binary classification task.

$$E = -\frac{1}{m} \sum [y_{\text{true}} * \log(y_{\text{pred}}) + (1 - y_{\text{true}}) * \log(1 - y_{\text{pred}})]$$

$$\frac{dE}{dy_{\text{pred}}} = -\left(\frac{y_{\text{true}}}{y_{\text{pred}}} - \frac{1 - y_{\text{true}}}{1 - y_{\text{pred}}}\right)$$

The Final Architecture: A Network to Classify Digits




```
# Load and preprocess MNIST data for '0' and '1'
x_train, y_train, x_test, y_test = preprocess_data(...)

# Define the network
network = [
    ConvolutionalLayer((1, 28, 28), kernel_size=3, depth=5),
    Sigmoid(),
    ReshapeLayer((5, 26, 26), (5 * 26 * 26, 1)),
    DenseLayer(5 * 26 * 26, 100),
    Sigmoid(),
    DenseLayer(100, 2),
    Sigmoid()
]

# Set loss function
loss = binary_cross_entropy
loss_prime = binary_cross_entropy_prime

# Train the network...
```

The Payoff: Watching the Network Learn

By implementing the **forward and backward passes** correctly, we provide the network with the mechanics it needs to learn from data. The training loop iteratively adjusts the **weights** in our convolutional and dense layers, minimising the error.

```
epoch 1/20   error=0.251
epoch 2/20   error=0.113
epoch 3/20   error=0.074
. . .
epoch 18/20  error=0.015
epoch 19/20  error=0.013
epoch 20/20  error=0.012
```

Prediction: 0, Confidence: 99.9%

Prediction: 1, Confidence: 99.8%

Prediction: 0, Confidence: 99.7%

We have not just used a tool; we have built the engine from scratch. This fundamental understanding is the key to innovating and moving beyond the limits of off-the-shelf solutions.