For Single Image

1. Importing Libraries

What each one does:

PIL (Pillow): Image helps you load, open, resize, and convert images.

NumPy (np): Converts the loaded image into a numeric array. CNN operations (convolution, pooling) rely on NumPy arrays.

Matplotlib (plt): Used to visualize images, feature maps, filters, etc.

In [1]:
```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
```

2. Set the Image Path

In [2]:
```
IMAGE_PATH = "image.jpg"
```

3. Load the Image (ensure RGB format)

What this does:

Image.open() Reads the file and loads it as a PIL image object.

.convert("RGB") Ensures the image has 3 channels (Red, Green, Blue).

Good for consistency, especially if your image is grayscale or RGBA.

Result: img is still a PIL image, not a NumPy array yet.

In [3]:
```
img = Image.open(IMAGE_PATH).convert("RGB")
```

4. (Optional) Resize the image

What this does:

Resizes your image to 128×128 pixels.

Useful when:

Your original image is too big.

You want consistent shape for convolution operations.

It's commented out so it does nothing until you remove the #.

```
In [4]: img = img.resize((128, 128))
```

5. Convert to NumPy Array

Purpose:

Converts PIL image → a NumPy array.

CNN operations need numeric arrays (pixels).

After this:

arr.shape gives image height × width × channels.

Pixel values range from 0 to 255 (uint8).

```
In [5]: arr = np.array(img)
```

6. Print Basic Information

What each tells you:

Shape:

(height, width, channels)

Example: (512, 512, 3)

Channels = 3 for RGB.

Dtype:

Usually uint8 (pixel values 0–255).

min/max:

The lowest and highest pixel intensities.

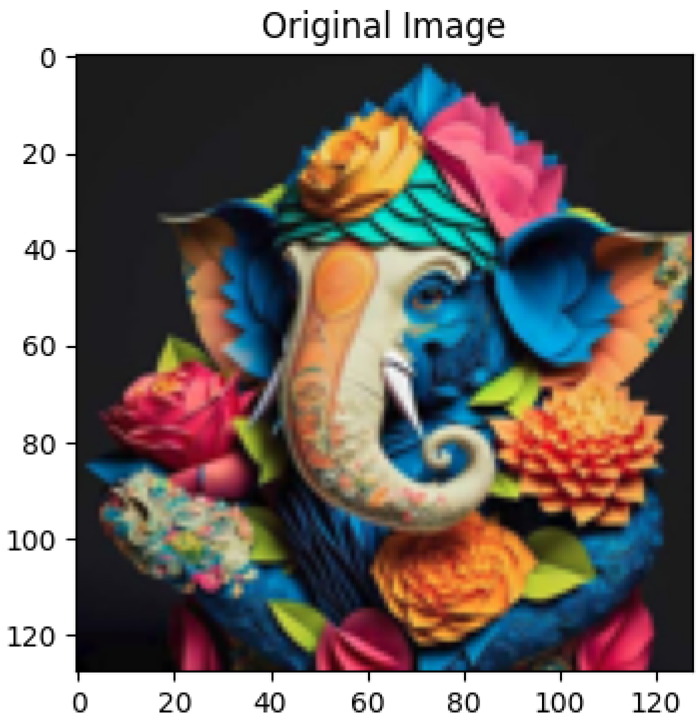
Helps verify the image loaded correctly.

```
In [6]: print("Image shape (H, W, C):", arr.shape)
        print("Dtype:", arr.dtype, "min/max:", arr.min(), arr.max())
```

```
Image shape (H, W, C): (128, 128, 3)
Dtype: uint8 min/max: 0 255
```

7. Display the Image

```
In [8]: plt.figure(figsize=(4,4))
        plt.imshow(arr)
plt.axis("on")
plt.title("Original Image")
plt.show()
```



Step-by-step:

plt.figure(figsize=(4,4))

Sets the display size of the plot.

plt.imshow(arr)

Displays the RGB image stored in the array.

plt.axis("off")

Removes axis ticks and labels for a cleaner look.

plt.title("Original Image")

Adds a title.

plt.show()

Renders the image.

Step 2 — Apply a 3×3 Edge-Detection Filter to Your Image Objective:

Create a simple 3×3 filter (kernel).

Slide it over your image.

Generate the feature map (output of convolution).

Visualize the result.

You will understand how CNN sees edges and patterns.

2A. Add this to a new cell in your notebook

1. Define a simple convolution function

This function works on one channel (grayscale). We will convert your image to grayscale first to keep things simple.

In [9]:
```python
def convolve2d(image, kernel):
    """
    image: 2D array (H, W)
    kernel: 2D array (k, k)
    returns: 2D convolved feature map
    """
    h, w = image.shape
    kh, kw = kernel.shape

    # Output size will be smaller
    output_h = h - kh + 1
    output_w = w - kw + 1

    output = np.zeros((output_h, output_w))

    # Perform convolution
    for i in range(output_h):
        for j in range(output_w):
            region = image[i:i+kh, j:j+kw]
            output[i, j] = np.sum(region * kernel)

    return output
```

2B. Convert your loaded image to grayscale

In [10]:
```python
gray = np.mean(arr, axis=2)  # average over RGB channels
plt.imshow(gray, cmap='gray')
plt.title("Grayscale Image")
plt.axis("off")
plt.show()
```

Grayscale Image

This gives you a simple 2D image.

2C. Define an edge-detection kernel (3×3)

This is a classic CNN starter filter.

```
In [11]: edge_kernel = np.array([
             [-1, -1, -1],
             [ 0,  0,  0],
             [ 1,  1,  1]
         ])
```

This filter detects horizontal edges.

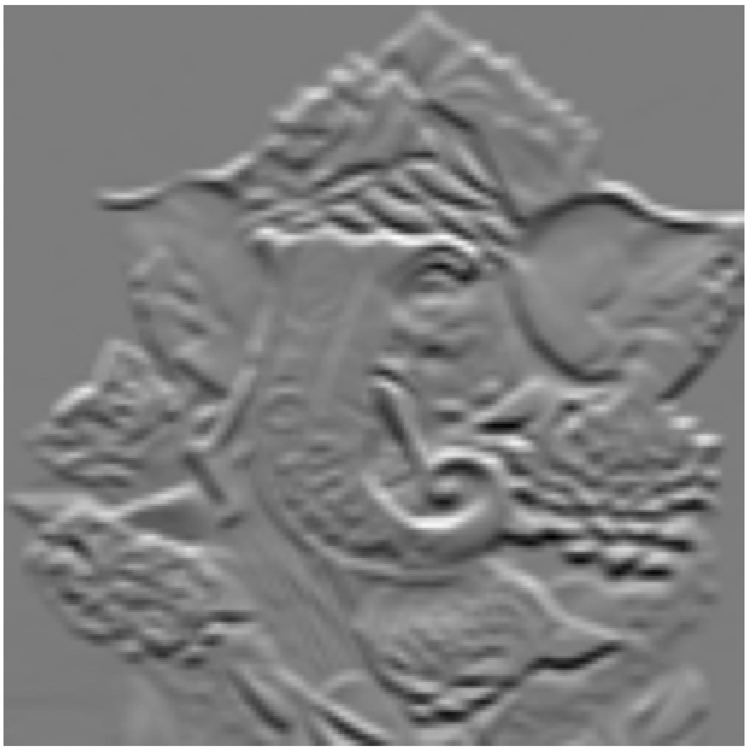
Later, we can do vertical, diagonal, blur, sharpen, etc.

2D. Apply convolution

```
In [14]: feature_map = convolve2d(gray, edge_kernel)

         plt.imshow(feature_map, cmap='gray')
         plt.title("Feature Map (After Convolution)")
         plt.axis("off")
         plt.show()
```

Feature Map (After Convolution)



Expected Result

You should see:

The image transformed into an edge-highlighted version.

Dark and bright lines where horizontal features exist.

A smaller output image (because of kernel size).

Step 3 — Apply ReLU Activation

What ReLU does

ReLU = Rectified Linear Unit Mathematically:

If value < 0 → make it 0

If value ≥ 0 → keep it

ReLU removes negative responses and keeps only strong positive activations.

3A. Define ReLU function

Add this in a new cell:

In [15]:
```
def relu(x):
    return np.maximum(0, x)
```

This uses NumPy’s built-in vectorized max function.

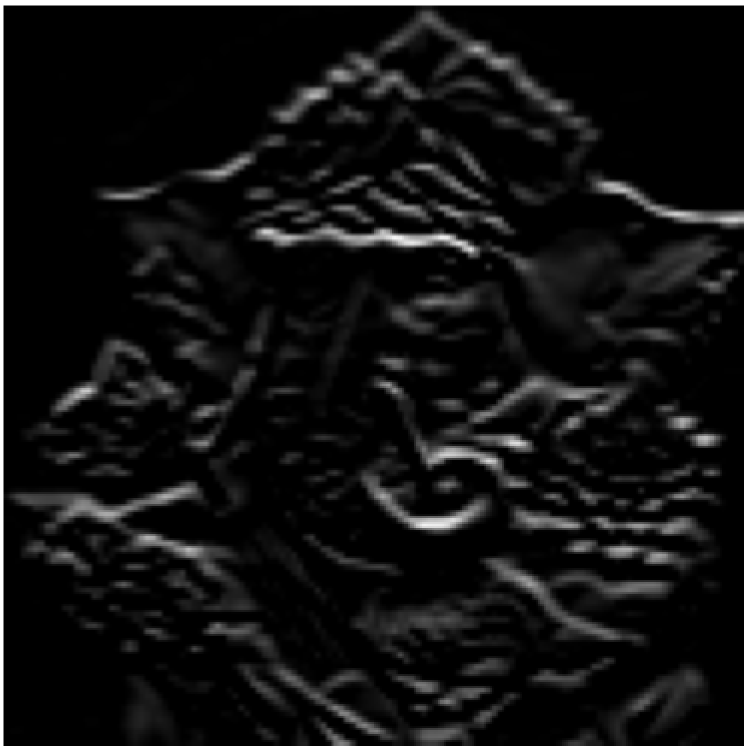
3B. Apply ReLU to your feature map

In [16]:
```
relu_feature = relu(feature_map)

plt.imshow(relu_feature, cmap='gray')
plt.title("After ReLU Activation")
plt.axis("off")
plt.show()

print("Before ReLU -> min:", feature_map.min(), "max:", feature_map.max())
print("After  ReLU -> min:", relu_feature.min(), "max:", relu_feature.max())
```

After ReLU Activation



```
Before ReLU -> min: -500.66666666666674 max: 523.3333333333333
After  ReLU -> min: 0.0 max: 523.3333333333333
```

Expected Output

The image becomes brighter where edges exist.

Negative values disappear (become 0).

Only positive activations remain.

You should observe:

Feature map min value ≤ 0

ReLU feature map min = 0

Step 4 — Apply 2×2 Max Pooling

Why we do pooling:

Reduce feature-map size

Keep only the most important values

Reduce computation

Make features more robust

Max Pooling (2×2) looks at each 2×2 block and selects the maximum value.

4A. Define a max pooling function

Add this to a new cell:

In [17]:
```
def max_pooling(feature_map, size=2):
    h, w = feature_map.shape
    new_h = h // size
    new_w = w // size

    pooled = np.zeros((new_h, new_w))

    for i in range(0, h, size):
        for j in range(0, w, size):
            block = feature_map[i:i+size, j:j+size]
            pooled[i//size, j//size] = np.max(block)

    return pooled
```

4B. Apply max pooling to your ReLU output

In [18]:
```
pooled_feature = max_pooling(relu_feature, size=2)

plt.imshow(pooled_feature, cmap='gray')
plt.title("After Max Pooling (2×2)")
plt.axis("off")
plt.show()

print("Original feature map shape:", relu_feature.shape)
print("Pooled feature map shape:", pooled_feature.shape)
```

After Max Pooling (2×2)

```
Original feature map shape: (126, 126)
Pooled feature map shape: (63, 63)
```

Expected Output

The image becomes smaller (roughly half height and half width).

Only the strongest pixels survive.

You will see a compressed version of the original feature map.

Step 5 — Flatten the pooled feature map

This step converts a 2D array → 1D vector, preparing it for a Dense layer.

This is what real CNNs do before classification.

Add this to a new cell:

In [19]:
```python
flattened = pooled_feature.flatten()

print("Flattened vector length:", len(flattened))
print(flattened[:20])  # show first 20 values
```

```
Flattened vector length: 3969
[0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Expected Output

A long 1D array of numbers

Length = height × width of pooled feature map

Values are positive numbers because we applied ReLU before pooling