Exercise 2: Refresh Computer Vision

1. Load the same image using OpenCV and PIL. • Print the shape, dtype, and value range for ea
   Explain any differences you observe.
2. Convert both images to grayscale. • Verify whether the outputs are numerically identical. • If
   explain why.
3. Read a color image using OpenCV and display it using matplotlib. • Why do the colors look
   incorrect? • Fix the issue and explain the root cause.
4. Split an RGB image into individual channels. • Visualize each channel. • Which objects appea
   brightest in each channel and why?
5. Convert an image from RGB to HSV. • Identify pixels corresponding to a specific color (e.g., r
   Explain why HSV is preferable to RGB for this task.
6. Resize an image to 224×224. • Identify geometric distortions. • Propose a method to preser
   ratio.
7. Normalize an image to the range [0,1]. • What happens if normalization is applied twice? • H
   this affect visualization

a. Create two random 3x3 tensors and perform matrix multiplication. Comput
the matrix
product and use Py Torch's autograd to calculate the gradient of the resul
with respect
to one of the input tensors

In [4]:
```
import torch

# Step 1: Create two random 3x3 tensors
A = torch.randn(3, 3, requires_grad=True)   # requires_grad=True so we can compu
B = torch.randn(3, 3)

print("Tensor A:\n", A)
print("Tensor B:\n", B)

# Step 2: Perform matrix multiplication
C = torch.matmul(A, B)
print("Matrix product C:\n", C)

# Step 3: Define a scalar output (needed for autograd)
# For example, take the sum of all elements in C
loss = C.sum()

# Step 4: Backpropagate to compute gradients
loss.backward()

# Step 5: Gradient of loss w.r.t. A
print("Gradient of loss w.r.t A:\n", A.grad)
```

```
Tensor A:
 tensor([[-1.0381, -0.7621, -1.5094],
        [ 0.7404,  1.0457, -0.0514],
        [ 0.6395, -0.8402,  0.6587]], requires_grad=True)
Tensor B:
 tensor([[-0.8843,  1.7355,  0.2425],
        [ 0.3276, -1.2903,  0.9603],
        [-0.0389,  0.4359, -1.0522]])
Matrix product C:
 tensor([[ 0.7270, -1.4762,  0.6046],
        [-0.3101, -0.0867,  1.2378],
        [-0.8664,  2.4810, -1.3448]], grad_fn=<MmBackward0>)
Gradient of loss w.r.t A:
 tensor([[ 1.0936, -0.0024, -0.6551],
        [ 1.0936, -0.0024, -0.6551],
        [ 1.0936, -0.0024, -0.6551]])
```

a. Matrix Multiplication & Autograd Matrix multiplication is handled by torch.matmul() or the @
To use Autograd, we must set requires_grad=True.

In [1]:
```python
import torch

# Create tensors
A = torch.randn(3, 3, requires_grad=True)
B = torch.randn(3, 3)

# Matrix multiplication
C = torch.matmul(A, B)

# We need a scalar value to call backward()
external_grad = torch.ones_like(C)
C.backward(external_grad)

print(f"Gradient of C with respect to A:\n{A.grad}")
```

```
Gradient of C with respect to A:
tensor([[-0.1128,  1.0434, -1.0700],
        [-0.1128,  1.0434, -1.0700],
        [-0.1128,  1.0434, -1.0700]])
```

b. Broadcasting LogicBroadcasting allows you to perform operations on tensors of different shap
without manually duplicating data.3x1 Tensor + 1x3 Tensor: PyTorch "stretches" the 3x1 vertically
1x3 horizontally to create two 3x3 matrices before adding them.Math: If A is (3,1) and B is (1
A + B$ results in a (3,3) tensor where $C_{ij} = A_i + B_j$.

In [2]:
```python
t1 = torch.tensor([[1], [2], [3]]) # 3x1
t2 = torch.tensor([[10, 20, 30]])  # 1x3
t3 = torch.randn(3, 3)             # 3x3

result = (t1 + t2) * t3
print(f"Resulting Shape: {result.shape}")
```

```
Resulting Shape: torch.Size([3, 3])
```

c. Reshaping and SlicingReshaping changes the view of the data without moving it in memory, pr
the total number of elements remains the same ($6 \times 4 = 24$ and $3 \times 8 = 24$).Resha
tensor.view(3, 8) or tensor.reshape(3, 8).Slicing: tensor[:, :2] selects all rows (represented by :) and
two columns (indices 0 and 1).

In [8]:
```
import torch

# 1. Create a 2D tensor of shape (6, 4)
# Using torch.arange to make the numbers easy to track
tensor_6x4 = torch.arange(24).view(6, 4)
print("Original Tensor (6x4):\n", tensor_6x4)

# 2. Reshape it into (3, 8)
tensor_3x8 = tensor_6x4.view(3, 8)
print("\nReshaped Tensor (3, 8):\n", tensor_3x8)

# 3. Extract specific slices
# Slicing syntax: [row_start:row_end, col_start:col_end]
# ":" means select all; ":2" means select from index 0 up to (but not including)
sliced_tensor = tensor_3x8[:, :2]
print("\nSliced Tensor (All rows, first 2 columns):\n", sliced_tensor)
```

```
Original Tensor (6x4):
 tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]])

Reshaped Tensor (3, 8):
 tensor([[ 0,  1,  2,  3,  4,  5,  6,  7],
        [ 8,  9, 10, 11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20, 21, 22, 23]])

Sliced Tensor (All rows, first 2 columns):
 tensor([[ 0,  1],
        [ 8,  9],
        [16, 17]])
```

d. NumPy Interoperability PyTorch and NumPy share the same underlying memory when convert
the CPU, making the transition very efficient.

In [7]:
```
import numpy as np

# NumPy to Torch
arr = np.array([1.0, 2.0, 3.0])
t = torch.from_numpy(arr)

# Operation
t = t * 10

# Torch to NumPy
final_arr = t.numpy()
```

e. Distributions and StatsThis exercise tests your ability to initialize specific distributions and perf
reduction operations.DistributionFunctionParametersUniformtorch.rand(5, 5)Range [0,

1)Normaltorch.randn(5, 5)Mean 0, Std Dev 1Process:Element-wise Multiply: Use the * operator.Sta
result.mean() and result.std().Flatten: result.view(-1) or result.flatten().Sum: result.sum().

In [9]:
```python
# 1. Uniform Distribution [0, 1)
uniform_tensor = torch.rand(5, 5)

# 2. Normal Distribution (Mean=0, Std=1)
normal_tensor = torch.randn(5, 5)

# 3. Element-wise multiplication
# Note: This is different from Matrix Multiplication (@)
product = uniform_tensor * normal_tensor

# 4. Compute Mean and Standard Deviation
# Using .item() converts a single-value tensor to a Python number
mean_val = product.mean()
std_val = product.std()

print(f"Product Mean: {mean_val.item():.4f}")
print(f"Product Std Dev: {std_val.item():.4f}")

# 5. Reshape into 1D tensor of size 25
# -1 tells PyTorch to "infer" the dimension based on the remaining elements
flattened = product.view(-1)
print(f"\nFlattened shape: {flattened.shape}")

# 6. Compute sum of all elements
total_sum = flattened.sum()
print(f"Sum of all elements: {total_sum.item():.4f}")
```

```
Product Mean: 0.1173
Product Std Dev: 0.4746

Flattened shape: torch.Size([25])
Sum of all elements: 2.9323
```