CV Practical - 4

Name: Ritu Pal

Enrollment No.: 230297

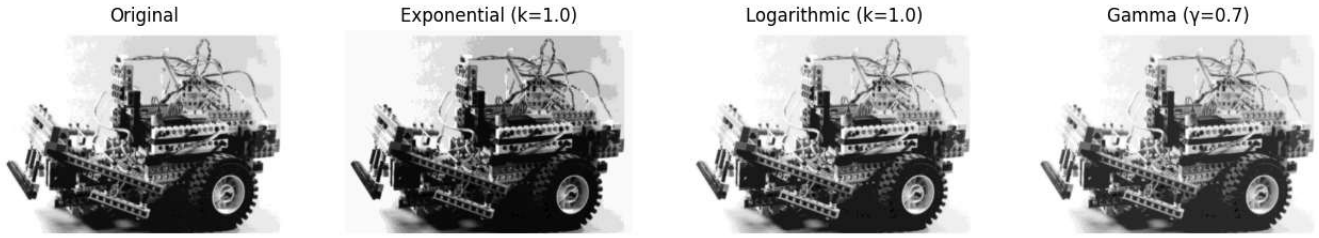
Batch: A3

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import files
import os

# ===== Upload images =====
uploaded = files.upload()  # upload your leg1 and trn1 here
image_paths = list(uploaded.keys())

# --------- Helpers ----------
def img_read_rgb(path):
    img = cv2.imread(path, cv2.IMREAD_COLOR)
    if img is None:
        raise FileNotFoundError(f"Cannot read image: {path}")
    return cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

def normalize01(im):
    im = im.astype(np.float32)
    im_min, im_max = im.min(), im.max()
    if im_max - im_min == 0:
        return np.zeros_like(im)
    return (im - im_min) / (im_max - im_min)

def to_uint8(im):
    return np.clip(im * 255.0, 0, 255).astype(np.uint8)

def entropy_gray(gray):
    hist = np.bincount((gray.flatten()).astype(np.uint8), minlength=256).astype(np.float64)
    p = hist / hist.sum()
    p_nonzero = p[p > 0]
    return -np.sum(p_nonzero * np.log2(p_nonzero))

def mean_gradient(gray):
    gx = cv2.Sobel(gray, cv2.CV_32F, 1, 0, ksize=3)
    gy = cv2.Sobel(gray, cv2.CV_32F, 0, 1, ksize=3)
    mag = np.sqrt(gx**2 + gy**2)
    return float(np.mean(mag))

# --------- Operators ----------
def exponential_operator(img01, k=1.0):
    denom = np.expm1(k)
    c = 1.0 / denom if denom != 0 else 1.0
    return np.clip(c * np.expm1(k * img01), 0.0, 1.0)

def logarithmic_operator(img01, k=1.0):
    denom = np.log1p(k)
    c = 1.0 / denom if denom != 0 else 1.0
    return np.clip(c * np.log1p(k * img01), 0.0, 1.0)

def gamma_correction(img01, gamma=1.0):
    return np.clip(img01 ** max(gamma, 1e-6), 0.0, 1.0)

# --------- Processing ----------
def process_and_compare(img_path, params):
    rgb = img_read_rgb(img_path)
    img01 = normalize01(rgb / 255.0)

    exp_img01 = exponential_operator(img01, k=params.get("k_exp", 1.0))
    log_img01 = logarithmic_operator(img01, k=params.get("k_log", 1.0))
    gamma_img01 = gamma_correction(img01, gamma=params.ge✦amma", 0.7))

    exp_rgb, log_rgb, gamma_rgb = map(to_uint8, [exp_img01, log_img01, gamma_img01])
```

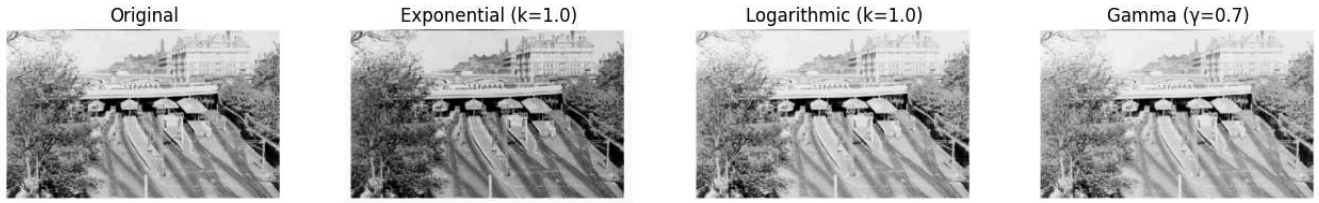
```
    gray_imgs = {
        "orig": cv2.cvtColor(to_uint8(img01), cv2.COLOR_RGB2GRAY),
        "exp": cv2.cvtColor(exp_rgb, cv2.COLOR_RGB2GRAY),
        "log": cv2.cvtColor(log_rgb, cv2.COLOR_RGB2GRAY),
        "gamma": cv2.cvtColor(gamma_rgb, cv2.COLOR_RGB2GRAY),
    }

    # Compute metrics
    results = {
        k: {
            "std": float(np.std(g)),
            "entropy": entropy_gray(g),
            "mean_grad": mean_gradient(g),
        }
        for k, g in gray_imgs.items()
    }

    # Show results
    fig, axes = plt.subplots(1, 4, figsize=(16,5))
    axes[0].imshow(to_uint8(img01)); axes[0].set_title("Original")
    axes[1].imshow(exp_rgb); axes[1].set_title(f"Exponential (k={params.get('k_exp',1.0)})")
    axes[2].imshow(log_rgb); axes[2].set_title(f"Logarithmic (k={params.get('k_log',1.0)})")
    axes[3].imshow(gamma_rgb); axes[3].set_title(f"Gamma (γ={params.get('gamma',0.7)})")
    for ax in axes: ax.axis('off')
    plt.show()

    return results

# ===== Run on uploaded images =====
params = {"k_exp": 1.0, "k_log": 1.0, "gamma": 0.7}

all_results = {}
for img_path in image_paths:
    print(f"\nProcessing: {img_path}")
    res = process_and_compare(img_path, params)
    all_results[img_path] = res
    for k,v in res.items():
        print(f"{k:>6} | std={v['std']:.2f} | entropy={v['entropy']:.3f} | mean_grad={v['mean_grad']:.3f}")
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Screenshot 2025-08-26 103433.png to Screenshot 2025-08-26 103433.png
Saving Screenshot 2025-08-26 103517.png to Screenshot 2025-08-26 103517.png

Processing: Screenshot 2025-08-26 103433.png
```




```
  orig | std=91.64 | entropy=5.330 | mean_grad=79.558
   exp | std=95.61 | entropy=5.278 | mean_grad=78.942
   log | std=87.71 | entropy=5.307 | mean_grad=78.187
 gamma | std=81.35 | entropy=5.264 | mean_grad=72.605

Processing: Screenshot 2025-08-26 103517.png
```

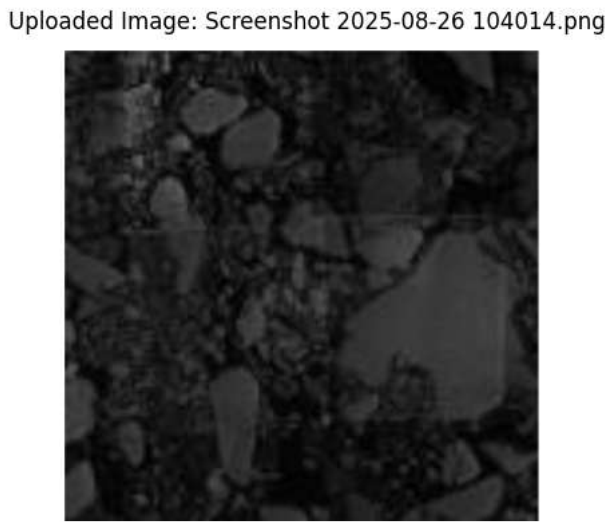



```
  orig | std=62.24 | entropy=7.346 | mean_grad=94.885
   exp | std=67.39 | entropy=7.248 | mean_grad=97.390
   log | std=58.13 | entropy=7.144 | mean_grad=90.339
 gamma | std=53.99 | entropy=7.023 | mean_grad=83.436
```

```
from google.colab import files
import cv2
import matplotlib.pyplot as plt

# ===== Upload image(s) =====
uploaded = files.upload()

# Get the first uploaded file name
filename = list(uploaded.keys())[0]
print("Using file:", filename)

# Read the uploaded file in grayscale
img = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
if img is None:
    raise FileNotFoundError(f"Could not read the uploaded image: {filename}")

# Show uploaded image
plt.imshow(img, cmap="gray")
plt.title(f"Uploaded Image: {filename}")
plt.axis("off")
plt.show()
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Screenshot 2025-08-26 104014.png to Screenshot 2025-08-26 104014.png
Using file: Screenshot 2025-08-26 104014.png
```

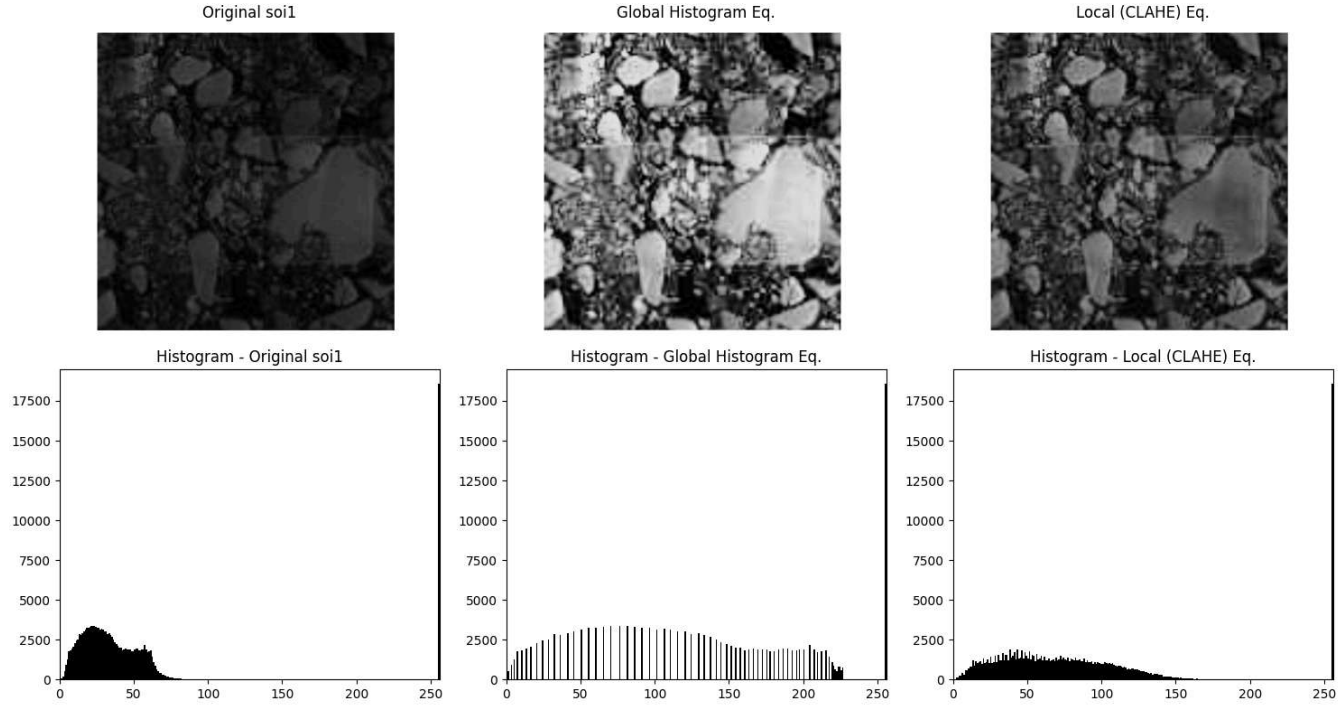



```
import cv2
import matplotlib.pyplot as plt
from google.colab import files

# Upload file
uploaded = files.upload()

# Get the actual filename from uploaded dict
filename = list(uploaded.keys())[0]

# Load image in grayscale
soi1 = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)

# Global Histogram Equalization
global_eq = cv2.equalizeHist(soi1)

# Local Histogram Equalization (CLAHE)
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
local_eq = clahe.apply(soi1)

# Titles and images
titles = ['Original soi1', 'Global Histogram Eq.', 'Local (CLAHE) Eq.']
images = [soi1, global_eq, local_eq]

plt.figure(figsize=(15, 8))

# Show images
```

```
for i in range(3):
    plt.subplot(2, 3, i+1)
    plt.imshow(images[i], cmap='gray')
    plt.title(titles[i])
    plt.axis('off')

# Show histograms
for i in range(3):
    plt.subplot(2, 3, i+4)
    plt.hist(images[i].ravel(), 256, [0,256], color='black')
    plt.title(f'Histogram - {titles[i]}')
    plt.xlim([0,256])

plt.tight_layout()
plt.show()
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Screenshot 2025-08-26 104014.png to Screenshot 2025-08-26 104014 (2).png
/tmp/ipython-input-3424027979.py:37: MatplotlibDeprecationWarning: Passing the range parameter of hist() positionally is deprecated
  plt.hist(images[i].ravel(), 256, [0,256], color='black')
```

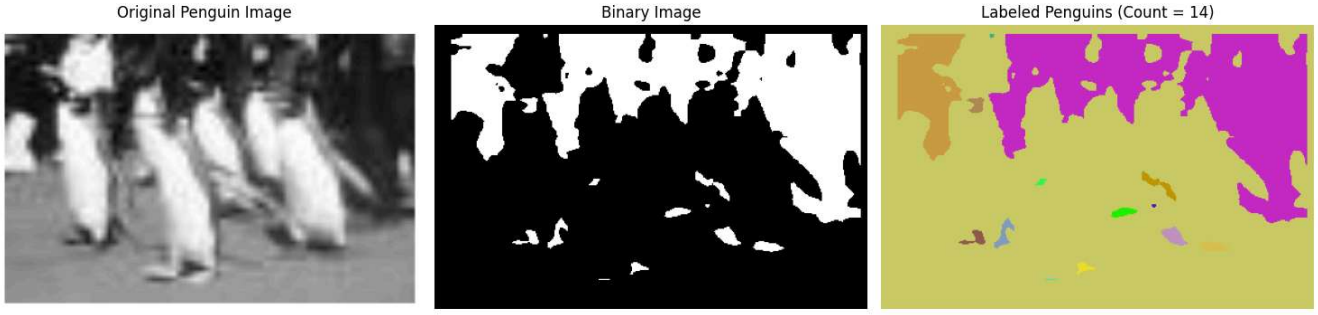



```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab import files

# --- Upload penguin image ---
uploaded = files.upload()
filename = list(uploaded.keys())[0]

# --- Load in grayscale ---
pen1 = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)

# --- Thresholding (binary image) ---
# Adjust threshold value (100) if penguins are not separated well
_, binary = cv2.threshold(pen1, 100, 255, cv2.THRESH_BINARY_INV)
```

```
# --- Connected Components ---
num_labels, labels = cv2.connectedComponents(binary)

# --- Random color mapping for visualization ---
label_img = np.zeros((labels.shape[0], labels.shape[1], 3), dtype=np.uint8)
colors = [tuple(np.random.randint(0,255,3).tolist()) for _ in range(num_labels)]

for r in range(labels.shape[0]):
    for c in range(labels.shape[1]):
        label_img[r, c] = colors[labels[r, c]]

# --- Show results ---
plt.figure(figsize=(15, 6))

plt.subplot(1, 3, 1)
plt.imshow(pen1, cmap='gray')
plt.title('Original Penguin Image')
plt.axis('off')

plt.subplot(1, 3, 2)
plt.imshow(binary, cmap='gray')
plt.title('Binary Image')
plt.axis('off')

plt.subplot(1, 3, 3)
plt.imshow(label_img)
plt.title(f'Labeled Penguins (Count = {num_labels-1})')
plt.axis('off')

plt.tight_layout()
plt.show()

print("Total penguins detected:", num_labels - 1)
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving Screenshot 2025-08-26 104312.png to Screenshot 2025-08-26 104312 (1).png
```




```
Total penguins detected: 14
```

CV Practical - 5

Name: Ritu Pal

Enrollment No.: 230297

Batch: A3

```
from scipy.ndimage import binary_hit_or_miss
```

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.morphology import skeletonize
```

Q1: Noise Reduction and Analysis

```
# ====== Step 1: Imports ======
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio as psnr

# ====== Step 2: Image Upload (For Google Colab) ======
from google.colab import files
uploaded = files.upload()

# Get uploaded image filename
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ====== Step 3: Q1 Function ======
def q1_noise_reduction(img_path):
    # Load grayscale image
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # Add salt-and-pepper noise
    noisy = img.copy()
    prob = 0.05  # noise probability
    rnd = np.random.rand(*img.shape)
    noisy[rnd < prob/2] = 0
    noisy[rnd > 1 - prob/2] = 255

    # Apply filters
    mean_f = cv2.blur(noisy, (3, 3))
    median_f = cv2.medianBlur(noisy, 3)
    gaussian_f = cv2.GaussianBlur(noisy, (3, 3), 0)

    # Compute PSNR
    psnr_mean = psnr(img, mean_f)
    psnr_median = psnr(img, median_f)
    psnr_gaussian = psnr(img, gaussian_f)

    # Display results
    titles = ["Original", "Noisy", "Mean Filter" ✦ edian Filter", "Gaussian Filter"]
    images = [img, noisy, mean_f, median_f, gaussian_f]

    plt.figure(figsize=(12,6))
```

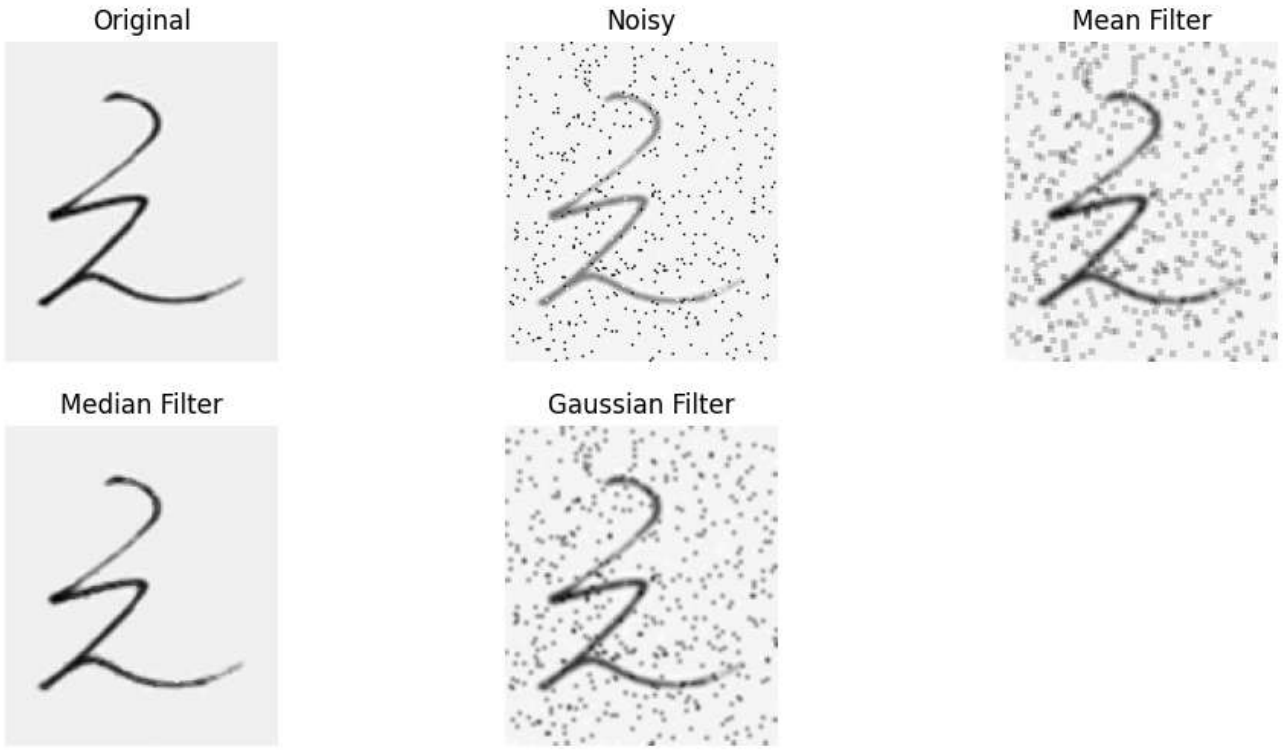
```
    for i in range(len(images)):
        plt.subplot(2,3,i+1)
        plt.imshow(images[i], cmap='gray')
        plt.title(titles[i])
        plt.axis("off")
    plt.show()

    print(f"PSNR (Mean Filter): {psnr_mean:.2f}")
    print(f"PSNR (Median Filter): {psnr_median:.2f}")
    print(f"PSNR (Gaussian Filter): {psnr_gaussian:.2f}")

# ====== Step 4: Run Q1 ======
q1_noise_reduction(img_path)
```

Choose files No file chosen

Upload widget is only available when the cell has been executed in the current browser session.

Please rerun this cell to enable.

```
Saving P5-1.jpg to P5-1.jpg
Uploaded: P5-1.jpg
```




```
PSNR (Mean Filter): 24.44
PSNR (Median Filter): 38.93
```

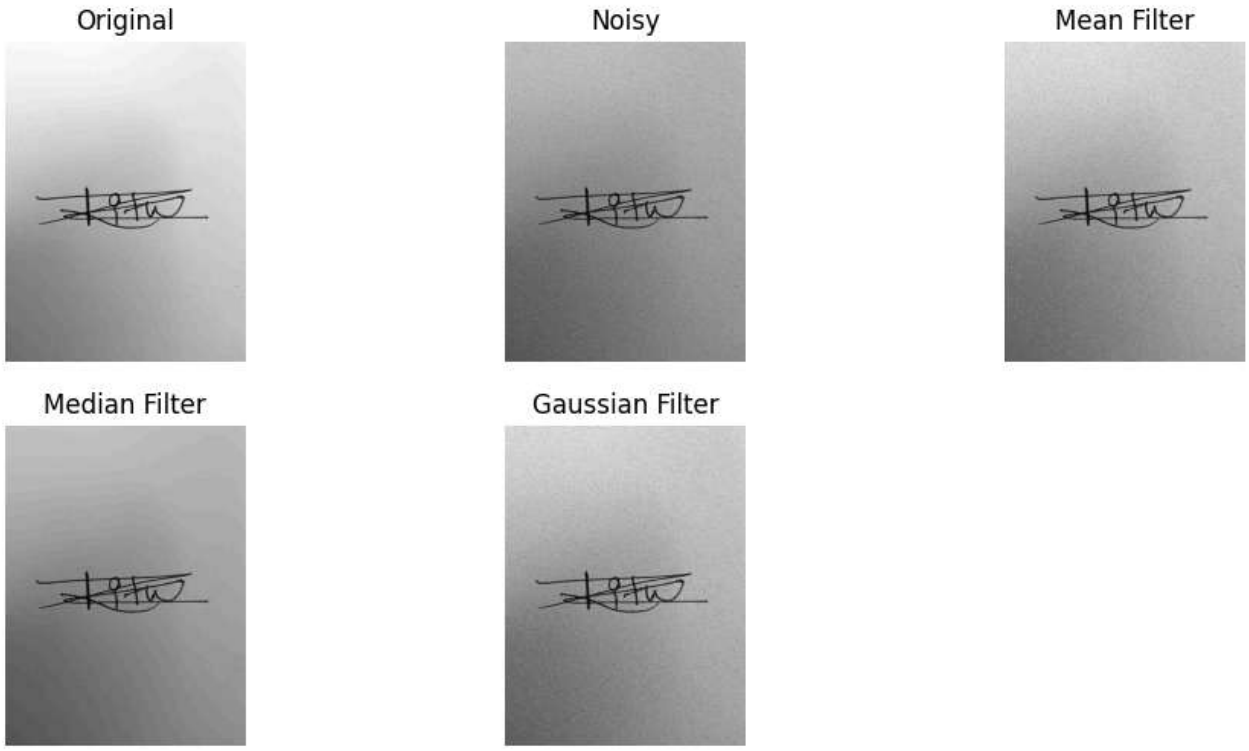
```
# Load grayscale or color image
img = cv2.imread(img_path)

# Print image shape
print("Image shape:", img.shape)

# If grayscale -> (height, width)
# If color -> (height, width, channels)

# Print width, height separately
h, w = img.shape[:2]
print("Width:", w, "Height:", h)
```

```
Image shape: (142, 121, 3)
Width: 121 Height: 142
```

```
# ====== Step 1: Imports ======
import cv2
```

```
import numpy as np
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio as psnr

# ====== Step 2: Image Upload (For Google Colab) ======
from google.colab import files
uploaded = files.upload()

# Get uploaded image filename
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ====== Step 3: Q1 Function ======
def q1_noise_reduction(img_path):
    # Load grayscale image
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # Add salt-and-pepper noise
    noisy = img.copy()
    prob = 0.05  # noise probability
    rnd = np.random.rand(*img.shape)
    noisy[rnd < prob/2] = 0
    noisy[rnd > 1 - prob/2] = 255

    # Apply filters
    mean_f = cv2.blur(noisy, (3, 3))
    median_f = cv2.medianBlur(noisy, 3)
    gaussian_f = cv2.GaussianBlur(noisy, (3, 3), 0)

    # Compute PSNR
    psnr_mean = psnr(img, mean_f)
    psnr_median = psnr(img, median_f)
    psnr_gaussian = psnr(img, gaussian_f)

    # Display results
    titles = ["Original", "Noisy", "Mean Filter", "Median Filter", "Gaussian Filter"]
    images = [img, noisy, mean_f, median_f, gaussian_f]

    plt.figure(figsize=(12,6))
    for i in range(len(images)):
        plt.subplot(2,3,i+1)
        plt.imshow(images[i], cmap='gray')
        plt.title(titles[i])
        plt.axis("off")
    plt.show()

    print(f"PSNR (Mean Filter): {psnr_mean:.2f}")
    print(f"PSNR (Median Filter): {psnr_median:.2f}")
    print(f"PSNR (Gaussian Filter): {psnr_gaussian:.2f}")

# ====== Step 4: Run Q1 ======
q1_noise_reduction(img_path)
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session.
Please rerun this cell to enable.
Saving signature.jpeg to signature.jpeg
Uploaded: signature.jpeg



PSNR (Mean Filter): 27.88
PSNR (Median Filter): 46.78

```
# Load grayscale or color image
img = cv2.imread(img_path)

# Print image shape
print("Image shape:", img.shape)

# If grayscale -> (height, width)
# If color -> (height, width, channels)

# Print width, height separately
h, w = img.shape[:2]
print("Width:", w, "Height:", h)
```

```
Image shape: (1600, 1200, 3)
Width: 1200 Height: 1600
```

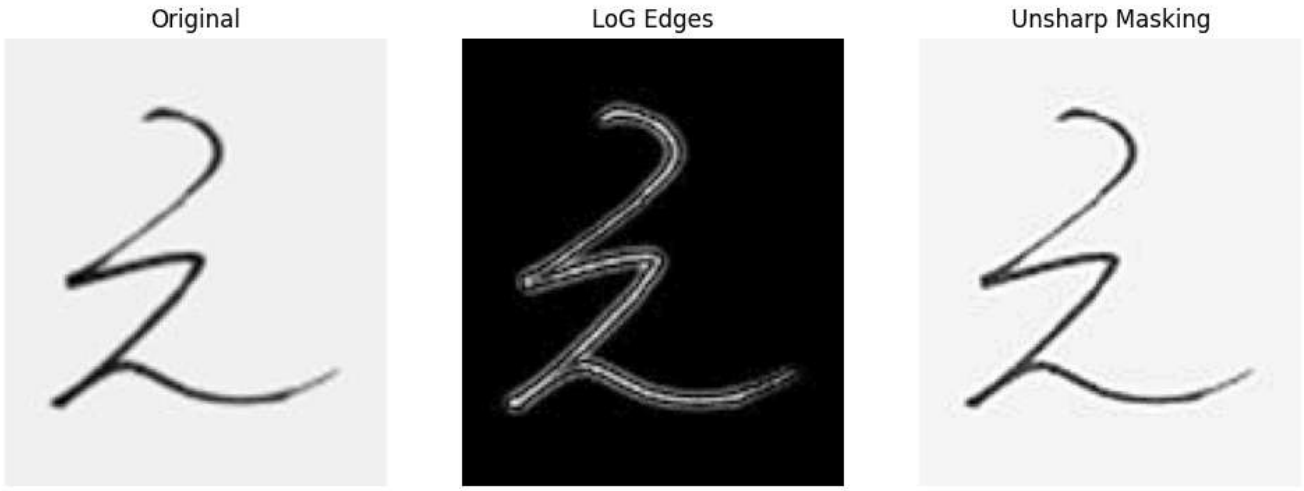
Q2: Edge Enhancement and Analysis

```
# ====== Step 1: Imports ======
import cv2
import matplotlib.pyplot as plt
from google.colab import files

# ====== Step 2: Upload Image ======
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ====== Step 3: Q2 Function ======
def q2_edge_enhancement(img_path, k=1.5):
```

```
    # Load grayscale image
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # (a) Laplacian of Gaussian (LoG)
    log = cv2.GaussianBlur(img, (3,3), 0)   # smooth first
    log = cv2.Laplacian(log, cv2.CV_64F)    # apply Laplacian
    log = cv2.convertScaleAbs(log)          # convert for display

    # (b) Unsharp Masking
    blur = cv2.GaussianBlur(img, (5,5), 0)
    unsharp = cv2.addWeighted(img, 1 + k, blur, -k, 0)   # formula

    # Display results
    plt.figure(figsize=(12,6))
    plt.subplot(1,3,1), plt.imshow(img, cmap='gray'), plt.title("Original")
    plt.axis("off")
    plt.subplot(1,3,2), plt.imshow(log, cmap='gray'), plt.title("LoG Edges")
    plt.axis("off")
    plt.subplot(1,3,3), plt.imshow(unsharp, cmap='gray'), plt.title("Unsharp Masking")
    plt.axis("off")
    plt.show()

# ====== Step 4: Run Q2 ======
q2_edge_enhancement(img_path, k=1.5)
```

```
Choose files No file chosen
Please rerun this cell to enable.
Saving P5-1.jpg to P5-1 (1).jpg
Uploaded: P5-1 (1).jpg
```

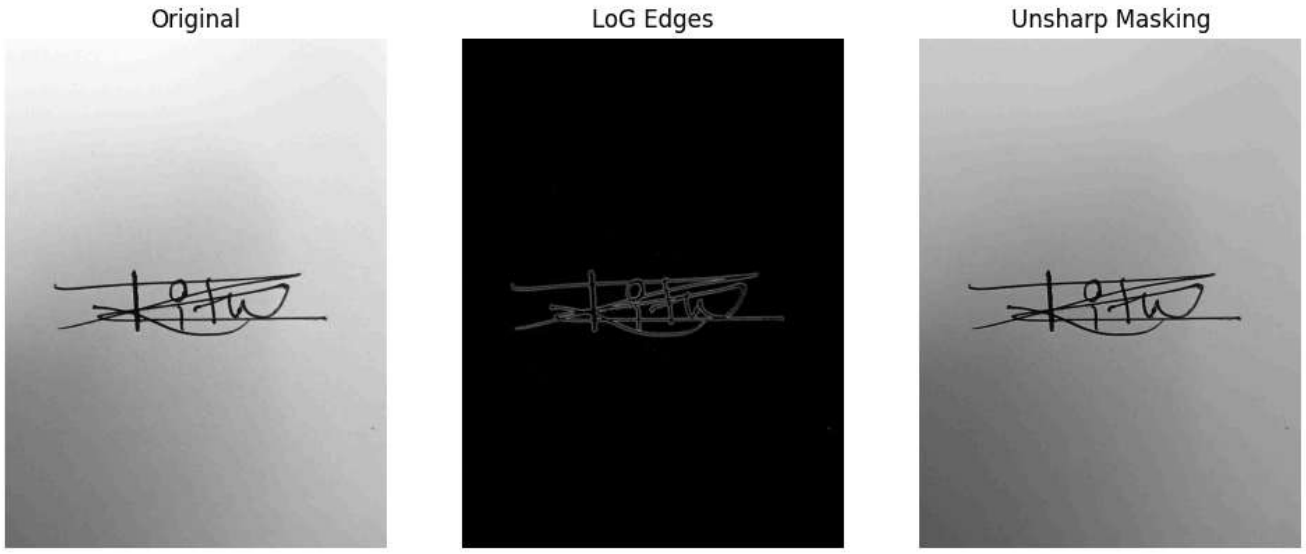
Upload widget is only available when the cell has been executed in the current browser session.




```
# ====== Step 1: Imports ======
import cv2
import matplotlib.pyplot as plt
from google.colab import files

# ====== Step 2: Upload Image ======
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ====== Step 3: Q2 Function ======
def q2_edge_enhancement(img_path, k=1.5):
    # Load grayscale image
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # (a) Laplacian of Gaussian (LoG)
    log = cv2.GaussianBlur(img, (3,3), 0)   # smooth first
    log = cv2.Laplacian(log, cv2.CV_64F)    # apply Laplacian
    log = cv2.convertScaleAbs(log)          # convert for display
```

```
    # (b) Unsharp Masking
    blur = cv2.GaussianBlur(img, (5,5), 0)
    unsharp = cv2.addWeighted(img, 1 + k, blur, -k, 0)   # formula

    # Display results
    plt.figure(figsize=(12,6))
    plt.subplot(1,3,1), plt.imshow(img, cmap='gray'), plt.title("Original")
    plt.axis("off")
    plt.subplot(1,3,2), plt.imshow(log, cmap='gray'), plt.title("LoG Edges")
    plt.axis("off")
    plt.subplot(1,3,3), plt.imshow(unsharp, cmap='gray'), plt.title("Unsharp Masking")
    plt.axis("off")
    plt.show()

# ====== Step 4: Run Q2 ======
q2_edge_enhancement(img_path, k=1.5)
```

```
Choose files No file chosen    Upload widget is only available when the cell has been executed in the current browser session.
Please rerun this cell to enable.
Saving signature.jpeg to signature (1).jpeg
Uploaded: signature (1).jpeg
```



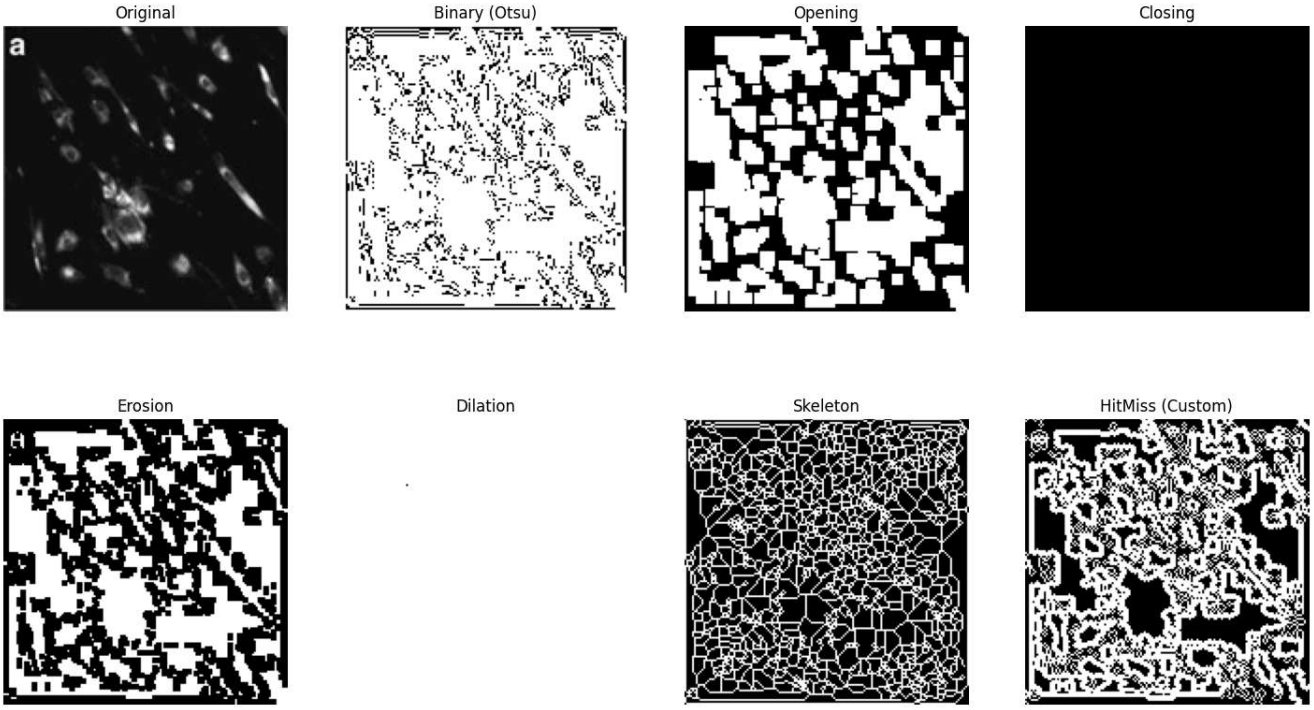
Q3. Cell Segmentation and Analysis in Microscopic Images

```
# ===== Step 1: Imports =====
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.morphology import skeletonize
from google.colab import files

# ===== Step 2: Upload Image =====
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ===== Step 3: Custom Hit-and-Miss Function =====
def hit_and_miss_custom(binary):
    results = []

    # Define circular kernels of different sizes
    kernels = [
        cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3)),
```

```
            cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5)),
            cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7,7))
        ]

        for k in kernels:
            # Erode with kernel (foreground match)
            fg = cv2.erode(binary, k)

            # Dilate to get background ring
            bg = cv2.dilate(fg, k)

            # Subtract to highlight matched circular pattern
            diff = cv2.subtract(bg, fg)
            results.append(diff)

        # Combine results from all kernel sizes
        combined = results[0]
        for r in results[1:]:
            combined = cv2.bitwise_or(combined, r)

        return combined

    # ===== Step 4: Morphological Operations Demo =====
    def morphological_demo(img_path):
        # Load grayscale image
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

        # Enhance contrast
        img_eq = cv2.equalizeHist(img)

        # Threshold (Otsu)
        _, binary = cv2.threshold(img_eq, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

        # Define kernel
        kernel = np.ones((3,3), np.uint8)

        # Opening: remove noise
        opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=2)

        # Closing: fill gaps
        closing = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel, iterations=2)

        # Erosion: shrink objects
        erosion = cv2.erode(binary, kernel, iterations=1)

        # Dilation: expand objects
        dilation = cv2.dilate(binary, kernel, iterations=1)

        # Skeletonization (medial axis)
        skeleton = skeletonize(binary > 0)

        # Custom Hit-and-Miss
        hitmiss_custom = hit_and_miss_custom(binary)

        # Display results
        titles = ["Original", "Binary (Otsu)", "Opening", "Closing",
                  "Erosion", "Dilation", "Skeleton", "HitMiss (Custom)"]
        images = [img, binary, opening, closing, erosion, dilation, skeleton, hitmiss_custom]

        plt.figure(figsize=(18,10))
        for i in range(len(images)):
            plt.subplot(2,4,i+1)
            plt.imshow(images[i], cmap="gray")
            plt.title(titles[i])
            plt.axis("off")
        plt.show()
```

```
# ===== Step 5: Run Demo =====
morphological_demo(img_path)
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session.
Please rerun this cell to enable.
Saving P5-2.jpg to P5-2.jpg
Uploaded: P5-2.jpg

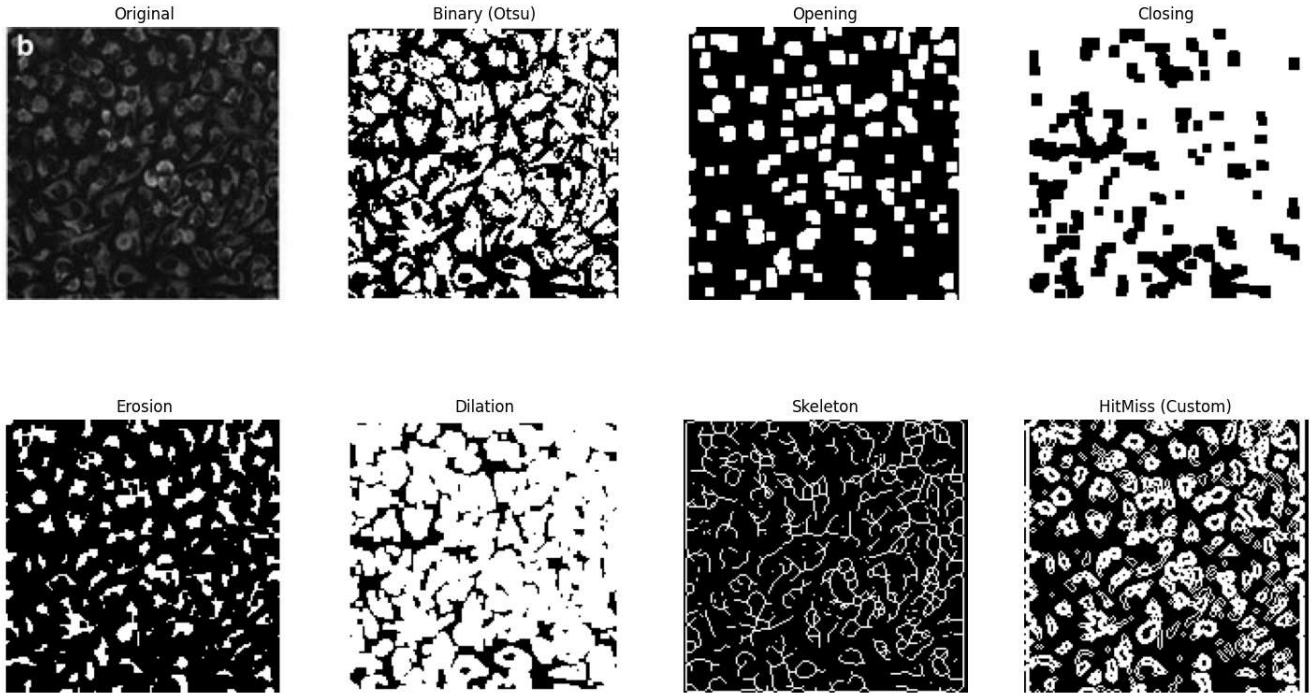



```
# ===== Step 1: Imports =====
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.morphology import skeletonize
from google.colab import files

# ===== Step 2: Upload Image =====
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ===== Step 3: Custom Hit-and-Miss Function =====
def hit_and_miss_custom(binary):
    results = []

    # Define circular kernels of different sizes
    kernels = [
```

```
            cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3)),
            cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5)),
            cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7,7))
        ]

        for k in kernels:
            # Erode with kernel (foreground match)
            fg = cv2.erode(binary, k)

            # Dilate to get background ring
            bg = cv2.dilate(fg, k)

            # Subtract to highlight matched circular pattern
            diff = cv2.subtract(bg, fg)
            results.append(diff)

        # Combine results from all kernel sizes
        combined = results[0]
        for r in results[1:]:
            combined = cv2.bitwise_or(combined, r)

        return combined

    # ===== Step 4: Morphological Operations Demo =====
    def morphological_demo(img_path):
        # Load grayscale image
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

        # Enhance contrast
        img_eq = cv2.equalizeHist(img)

        # Threshold (Otsu)
        _, binary = cv2.threshold(img_eq, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

        # Define kernel
        kernel = np.ones((3,3), np.uint8)

        # Opening: remove noise
        opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=2)

        # Closing: fill gaps
        closing = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel, iterations=2)

        # Erosion: shrink objects
        erosion = cv2.erode(binary, kernel, iterations=1)

        # Dilation: expand objects
        dilation = cv2.dilate(binary, kernel, iterations=1)

        # Skeletonization (medial axis)
        skeleton = skeletonize(binary > 0)

        # Custom Hit-and-Miss
        hitmiss_custom = hit_and_miss_custom(binary)

        # Display results
        titles = ["Original", "Binary (Otsu)", "Opening", "Closing",
                  "Erosion", "Dilation", "Skeleton", "HitMiss (Custom)"]
        images = [img, binary, opening, closing, erosion, dilation, skeleton, hitmiss_custom]

        plt.figure(figsize=(18,10))
        for i in range(len(images)):
            plt.subplot(2,4,i+1)
            plt.imshow(images[i], cmap="gray")
            plt.title(titles[i])
            plt.axis("off")
        plt.show()
```

```
# ===== Step 5: Run Demo =====
morphological_demo(img_path)
```

Choose files No file chosen
Upload widget is only available when the cell has been executed in the current browser session.
Please rerun this cell to enable.
Saving P5-3.jpg to P5-3.jpg
Uploaded: P5-3.jpg

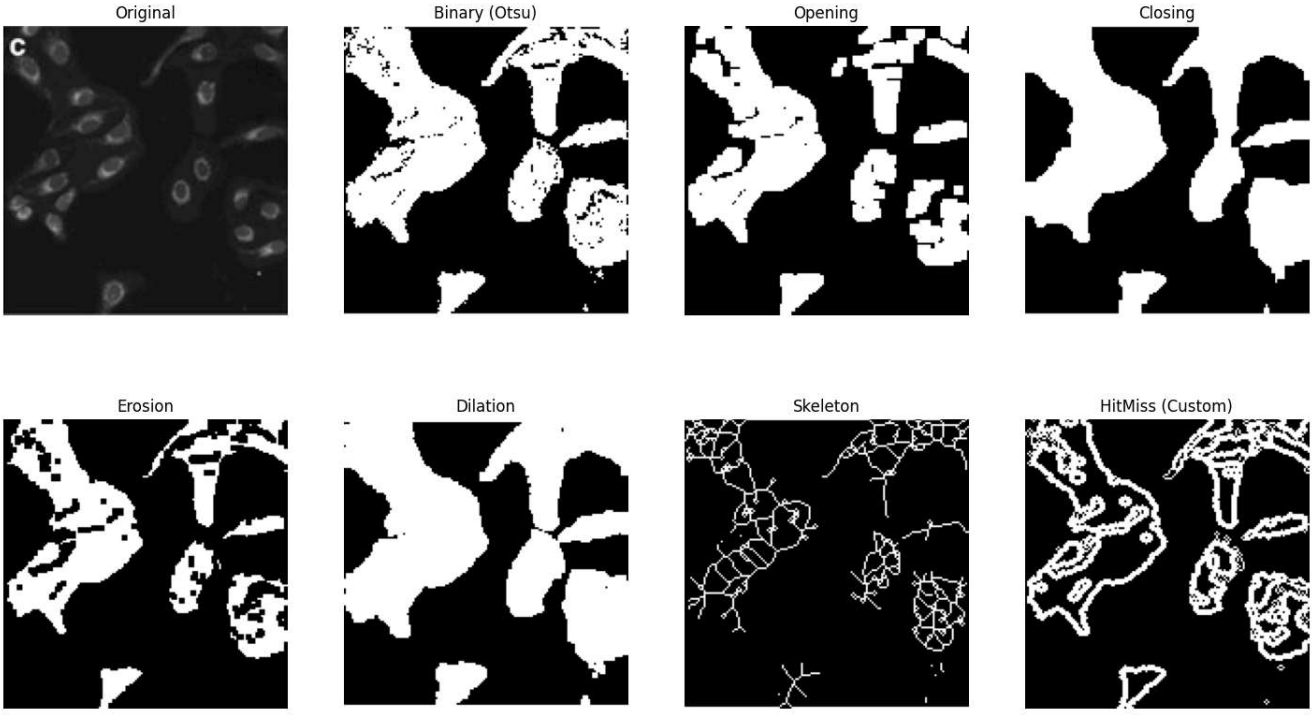



```
# ===== Step 1: Imports =====
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.morphology import skeletonize
from google.colab import files

# ===== Step 2: Upload Image =====
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ===== Step 3: Custom Hit-and-Miss Function =====
def hit_and_miss_custom(binary):
    results = []

    # Define circular kernels of different sizes
    kernels = [
        cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3)),
        cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5)),
```

```
        cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7,7))
    ]

    for k in kernels:
        # Erode with kernel (foreground match)
        fg = cv2.erode(binary, k)

        # Dilate to get background ring
        bg = cv2.dilate(fg, k)

        # Subtract to highlight matched circular pattern
        diff = cv2.subtract(bg, fg)
        results.append(diff)

    # Combine results from all kernel sizes
    combined = results[0]
    for r in results[1:]:
        combined = cv2.bitwise_or(combined, r)

    return combined

# ===== Step 4: Morphological Operations Demo =====
def morphological_demo(img_path):
    # Load grayscale image
    img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

    # Enhance contrast
    img_eq = cv2.equalizeHist(img)

    # Threshold (Otsu)
    _, binary = cv2.threshold(img_eq, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

    # Define kernel
    kernel = np.ones((3,3), np.uint8)

    # Opening: remove noise
    opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=2)

    # Closing: fill gaps
    closing = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel, iterations=2)

    # Erosion: shrink objects
    erosion = cv2.erode(binary, kernel, iterations=1)

    # Dilation: expand objects
    dilation = cv2.dilate(binary, kernel, iterations=1)

    # Skeletonization (medial axis)
    skeleton = skeletonize(binary > 0)

    # Custom Hit-and-Miss
    hitmiss_custom = hit_and_miss_custom(binary)

    # Display results
    titles = ["Original", "Binary (Otsu)", "Opening", "Closing",
              "Erosion", "Dilation", "Skeleton", "HitMiss (Custom)"]
    images = [img, binary, opening, closing, erosion, dilation, skeleton, hitmiss_custom]

    plt.figure(figsize=(18,10))
    for i in range(len(images)):
        plt.subplot(2,4,i+1)
        plt.imshow(images[i], cmap="gray")
        plt.title(titles[i])
        plt.axis("off")
    plt.show()
```

```
# ===== Step 5: Run Demo =====
morphological_demo(img_path)
```

Choose files No file chosen Upload widget is only available when the cell has been executed in the current browser session.
Please rerun this cell to enable.
Saving P5-4.jpg to P5-4.jpg
Uploaded: P5-4.jpg

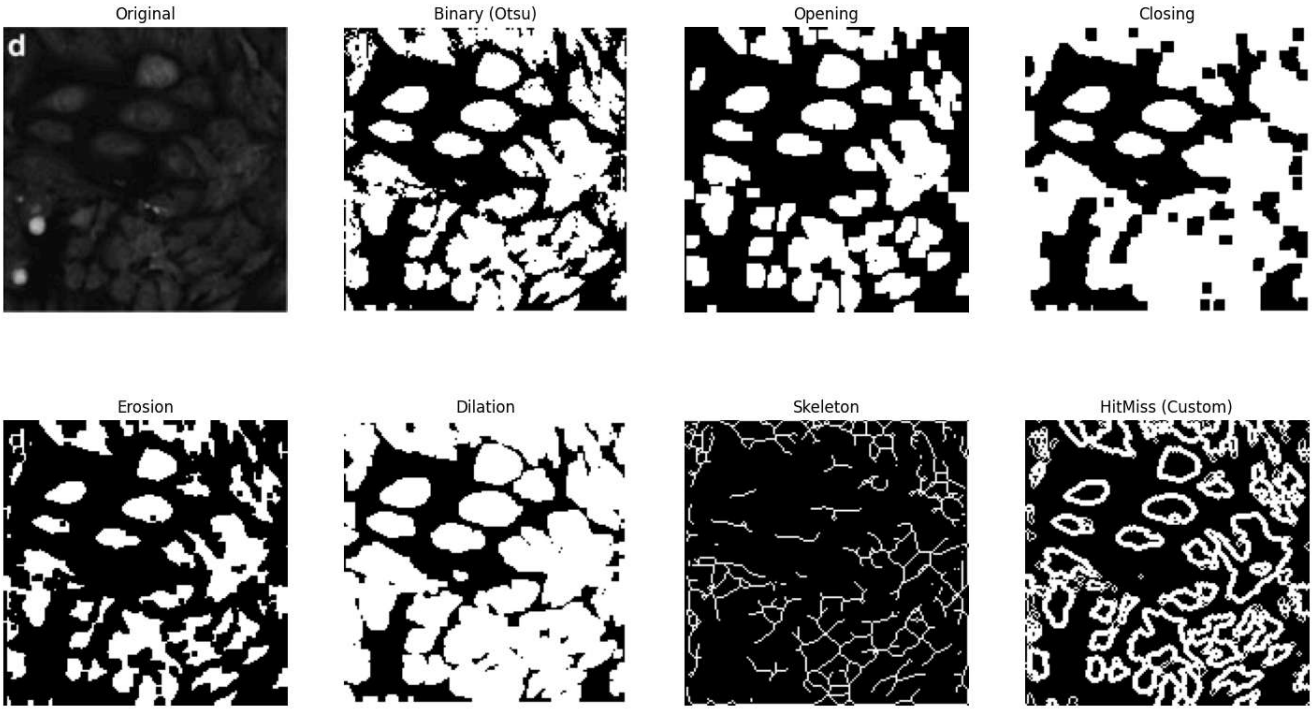



```
# ===== Step 1: Imports =====
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.morphology import skeletonize
from google.colab import files

# ===== Step 2: Upload Image =====
uploaded = files.upload()
img_path = list(uploaded.keys())[0]
print("Uploaded:", img_path)

# ===== Step 3: Custom Hit-and-Miss Function =====
def hit_and_miss_custom(binary):
    results = []

    # Define circular kernels of different sizes
    kernels = [
        cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3,3)),
        cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5)),
```

```
            cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (7,7))
        ]

        for k in kernels:
            # Erode with kernel (foreground match)
            fg = cv2.erode(binary, k)

            # Dilate to get background ring
            bg = cv2.dilate(fg, k)

            # Subtract to highlight matched circular pattern
            diff = cv2.subtract(bg, fg)
            results.append(diff)

        # Combine results from all kernel sizes
        combined = results[0]
        for r in results[1:]:
            combined = cv2.bitwise_or(combined, r)

        return combined

    # ===== Step 4: Morphological Operations Demo =====
    def morphological_demo(img_path):
        # Load grayscale image
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)

        # Enhance contrast
        img_eq = cv2.equalizeHist(img)

        # Threshold (Otsu)
        _, binary = cv2.threshold(img_eq, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)

        # Define kernel
        kernel = np.ones((3,3), np.uint8)

        # Opening: remove noise
        opening = cv2.morphologyEx(binary, cv2.MORPH_OPEN, kernel, iterations=2)

        # Closing: fill gaps
        closing = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel, iterations=2)

        # Erosion: shrink objects
        erosion = cv2.erode(binary, kernel, iterations=1)

        # Dilation: expand objects
        dilation = cv2.dilate(binary, kernel, iterations=1)

        # Skeletonization (medial axis)
        skeleton = skeletonize(binary > 0)

        # Custom Hit-and-Miss
        hitmiss_custom = hit_and_miss_custom(binary)

        # Display results
        titles = ["Original", "Binary (Otsu)", "Opening", "Closing",
                  "Erosion", "Dilation", "Skeleton", "HitMiss (Custom)"]
        images = [img, binary, opening, closing, erosion, dilation, skeleton, hitmiss_custom]

        plt.figure(figsize=(18,10))
        for i in range(len(images)):
            plt.subplot(2,4,i+1)
            plt.imshow(images[i], cmap="gray")
            plt.title(titles[i])
            plt.axis("off")
        plt.show()
```

Choose files No file chosen
Please rerun this cell to enable.
Saving P5-5.jpg to P5-5.jpg
Uploaded: P5-5.jpg

Upload widget is only available when the cell has been executed in the current browser session.



Thresholding Method	Before Processing	After Processing	Insight
Otsu Thresholding	1 cell	1 cell	Otsu used a **global threshold**, so only one very bright region was detected. The rest of the cells v
Adaptive Thresholding	1 cell	40 cells	Adaptive works on **local contrast** (per image region), so it correctly segmented many cells even i

Use Adaptive Thresholding for images with uneven illumination (like microscopy images).

Otsu is best for uniform lighting images or high-contrast datasets.

Morphological operations greatly improved segmentation accuracy by removing noise and consolidating true cell shapes.

CV Practical - 6: Part 1

Name: Ritu Pal

Enrollment No.: 230297

Batch: A3

Model Fitting & Optimization from Scratch (NumPy)

Goal: Implement a multi-layer neural network *from scratch* using NumPy to learn classifiers on MNIST and CIFAR. This Colab notebook guides students through writing activations, forward/backward passes, loss functions (cross-entropy, Huber/L1/L2), and implementing SGD with momentum and weight decay.

Notes for instructors:

- This notebook is structured with explanation cells and code cells. Students should fill the `TODO` sections.
- Designed to run in Google Colab. For CIFAR training you'll likely need GPU runtime to speed up.

---

1) Setup & Imports

Run the cell below to import libraries. If running in a fresh Colab environment, TensorFlow is available by default to load datasets. This notebook uses NumPy only for model math (no autograd).

```
# Standard imports - this code is ready for Colab
import numpy as np
import matplotlib.pyplot as plt
from time import time
from typing import List, Tuple, Dict, Any

# For dataset loading (Colab has tensorflow installed)
try:
    from tensorflow.keras.datasets import mnist, cifar10
    tf_available = True
except Exception as e:
    print("TensorFlow dataset loader not available. You can still load datasets manually.")
    tf_available = False

np.random.seed(42)
```

2) Utilities

Helper functions for one-hot encoding, stable softmax, accuracy and plotting training curves.

1. One-hot encoding Convert integer labels to vector form for loss computation
2. Stable Softmax Safely convert logits to probabilities without overflow
3. Accuracy Measure model correctness
4. Plot training curves Visualize learning progress & detect over/underfitting

```
def one_hot(labels: np.ndarray, num_classes: int) -> np.ndarray:
    y = np.zeros((labels.shape[0], num_classes), dtype=np.float32)
    for i, lab in enumerate(labels):
        y[i, int(lab)] = 1.0
    return y

def stable_softmax(x: np.ndarray) -> np.ndarray:
    # x: (batch, classes)
    x_max = np.max(x, axis=1, keepdims=True)
    ex = np.exp(x - x_max)
    return ex / np.sum(ex, axis=1, keepdims=True)
```

```
def accuracy(pred_probs: np.ndarray, labels: np.ndarray) -> float:
    preds = np.argmax(pred_probs, axis=1)
    return np.mean(preds == labels) * 100.0

def plot_training(history: Dict[str, List[float]]):
    epochs = len(history['train_loss'])
    fig, ax = plt.subplots(1,2, figsize=(12,4))
    ax[0].plot(range(1, epochs+1), history['train_loss'], label='train loss')
    ax[0].plot(range(1, epochs+1), history.get('val_loss', []), label='val loss')
    ax[0].set_xlabel('Epoch'); ax[0].set_ylabel('Loss'); ax[0].legend()
    ax[1].plot(range(1, epochs+1), history['train_acc'], label='train acc')
    ax[1].plot(range(1, epochs+1), history.get('val_acc', []), label='val acc')
    ax[1].set_xlabel('Epoch'); ax[1].set_ylabel('Accuracy (%)'); ax[1].legend()
    plt.show()

print('Utilities loaded.')
```

```
Utilities loaded.
```

3) Activations & Derivatives

Implement activations and their derivatives. Softmax is handled in the final layer; for backprop we provide a Jacobian helper for educational clarity (students may use alternative vectorized approaches). Educational Notes for Students

Sigmoid

Squashes values to (0, 1) range.

Good for probabilities but suffers from vanishing gradients for large |x|.

Derivative: $\sigma'(x)$

$\sigma(x)(1-\sigma(x))$ σ′(x)=σ(x)(1−σ(x))

Tanh

Output in (-1, 1).

Centered, but still has vanishing gradient issues.

Derivative: $1-\tanh^2(x)$ 1−tanh2(x)

ReLU

Zero for negative inputs, identity for positive.

Fast convergence but can cause "dead neurons".

Derivative: 0 if $x \le 0$ x≤0, else 1.

Leaky ReLU

Fixes dead neuron issue by giving small slope (α) for negatives.

Softmax

Converts logits to probabilities for multi-class classification.

In backprop, for cross-entropy + softmax, derivative simplifies to:

$$\frac{\partial L}{\partial z}$$

$\text{softmax}(z) - y$ one-hot ∂z ∂L

=softmax(z)−y one-hot

so in practice you don't need the full Jacobian.

```
def linear(x): return x
def d_linear(out): return np.ones_like(out)

def sigmoid(x): return 1.0 / (1.0 + np.exp(-x))
def d_sigmoid(out): return out * (1.0 - out)

def tanh_act(x): return np.tanh(x)
```

```
def d_tanh(out): return 1.0 - out**2

def relu(x): return np.maximum(0, x)
def d_relu(out): return (out > 0).astype(np.float32)

def leaky_relu(x, alpha=0.01):
    return np.where(x >= 0, x, alpha * x)
def d_leaky_relu_out(out, alpha=0.01):
    # out is activated output; derivative needs original x, but we can reconstruct sign from out for leaky relu
    # For positive outputs derivative is 1, for negative it's alpha (approximately)
    deriv = np.ones_like(out)
    deriv[out < 0] = alpha
    return deriv

# Softmax forward is stable_softmax above.
# For backward, we'll provide a Jacobian computation for a single data point (educational)
def softmax_jacobian(s: np.ndarray) -> np.ndarray:
    # s is 1D softmax output vector (C,)
    s = s.reshape(-1,1)
    return np.diagflat(s) - s.dot(s.T)

def d_softmax(prev_grad: np.ndarray, softmax_out: np.ndarray) -> np.ndarray:
    # prev_grad: (batch, classes) gradient coming from next layer (dL/dy)
    # softmax_out: (batch, classes)
    # Compute dL/dz where z is pre-softmax inputs.
    batch = prev_grad.shape[0]
    out = np.zeros_like(prev_grad)
    for i in range(batch):
        J = softmax_jacobian(softmax_out[i])
        out[i] = prev_grad[i].dot(J)
    return out

# Activation dispatcher
ACT_FNS = {
    'linear': (linear, d_linear),
    'sigmoid': (sigmoid, d_sigmoid),
    'tanh': (tanh_act, d_tanh),
    'relu': (relu, d_relu),
    'leaky_relu': (leaky_relu, d_leaky_relu_out),
}

print('Activations ready.')
```

```
Activations ready.
```

4) Loss Functions

Implement cross-entropy (with softmax), L1, L2, and Huber losses and their derivatives.

```
def cross_entropy_loss(probs: np.ndarray, targets: np.ndarray) -> float:
    # probs: (batch, classes) from softmax, targets: (batch, classes) one-hot
    # small epsilon for numerical stability
    eps = 1e-12
    probs_clipped = np.clip(probs, eps, 1. - eps)
    loss = -np.sum(targets * np.log(probs_clipped)) / probs.shape[0]
    return loss

def d_cross_entropy(probs: np.ndarray, targets: np.ndarray) -> np.ndarray:
    # derivative dL/dz where z is pre-softmax inputs if probs = softmax(z)
    # For cross-entropy with softmax, derivative simplifies to (probs - targets)/batch
    batch = probs.shape[0]
    return (probs - targets) / batch

def l2_loss(pred: np.ndarray, target: np.ndarray):
    diff = pred - target
    return 0.5 * np.mean(diff**2)

def d_l2(pred: np.ndarray, target: np.ndarray):
    batch = pred.shape[0]
    return (pred - target) / batch

def l1_loss(pred: np.ndarray, target: np.ndarray):
    return np.mean(np.abs(pred - target))
```

```
def d_l1(pred: np.ndarray, target: np.ndarray):
    diff = pred - target
    batch = pred.shape[0]
    # subgradient: sign(diff)
    return np.sign(diff) / batch

def huber_loss(pred: np.ndarray, target: np.ndarray, delta=1.0):
    diff = pred - target
    absd = np.abs(diff)
    mask = absd <= delta
    loss = np.where(mask, 0.5 * diff**2, delta * (absd - 0.5 * delta))
    return np.mean(loss)

def d_huber(pred: np.ndarray, target: np.ndarray, delta=1.0):
    diff = pred - target
    absd = np.abs(diff)
    mask = absd <= delta
    grad = np.where(mask, diff, delta * np.sign(diff))
    batch = pred.shape[0]
    return grad / batch

print('Loss functions ready.')
```

```
Loss functions ready.
```

5) Layer & Model Classes

Implement a simple Dense layer and Model class to handle forward/backward and parameter updates. Weight initialization options: Xavier (Glorot) and He.

```
class DenseLayer:
    def __init__(self, input_dim:int, output_dim:int, activation='relu', init='xavier'):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.activation_name = activation
        self.activation, self.d_activation = ACT_FNS.get(activation, ACT_FNS['relu'])

        # weight init
        if init == 'xavier':
            limit = np.sqrt(6.0 / (input_dim + output_dim))
            self.W = np.random.uniform(-limit, limit, size=(input_dim, output_dim)).astype(np.float32)
        elif init == 'he':
            std = np.sqrt(2.0 / input_dim)
            self.W = np.random.randn(input_dim, output_dim).astype(np.float32) * std
        else:
            self.W = np.random.randn(input_dim, output_dim).astype(np.float32) * 0.01

        self.b = np.zeros((1, output_dim), dtype=np.float32)
        # gradients and velocity for momentum
        self.dW = np.zeros_like(self.W)
        self.db = np.zeros_like(self.b)
        self.vW = np.zeros_like(self.W)
        self.vb = np.zeros_like(self.b)

        # caches for backprop
        self.x = None
        self.z = None
        self.a = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        # x: (batch, input_dim)
        self.x = x
        self.z = x.dot(self.W) + self.b  # (batch, output_dim)
        # apply activation (special-case softmax handled by Model)
        if self.activation_name == 'softmax':
            self.a = stable_softmax(self.z)
        else:
            # activation functions expect pre-activation input; some were implemented to accept x
            fn = ACT_FNS[self.activation_name][0]
            self.a = fn(self.z)
        return self.a

    def backward(self, grad_a: np.ndarray) -> np.ndarray:
        # grad_a: dL/da (batch, output_dim)
```

```
        # compute dL/dz first
        if self.activation_name == 'softmax':
            # use simplified derivative if combined with cross-entropy outside (handled elsewhere)
            # Otherwise use Jacobian-based derivative
            grad_z = d_softmax(grad_a, self.a)
        else:
            # derivative function expects activated output or z depending on implementation; we use a
            dfn = ACT_FNS[self.activation_name][1]
            grad_z = grad_a * dfn(self.a)

        # gradients w.r.t weights and bias
        batch = self.x.shape[0]
        self.dW = self.x.T.dot(grad_z)  # (input_dim, output_dim)
        self.db = np.sum(grad_z, axis=0, keepdims=True)
        # gradient w.r.t input to propagate to previous layer
        grad_x = grad_z.dot(self.W.T)
        return grad_x

class SimpleModel:
    def __init__(self, layers: List[DenseLayer]):
        self.layers = layers

    def forward(self, x: np.ndarray) -> np.ndarray:
        out = x
        for i, layer in enumerate(self.layers):
            out = layer.forward(out)
        return out

    def backward(self, grad_output: np.ndarray) -> None:
        grad = grad_output
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def update(self, lr: float, momentum: float=0.9, decay: float=0.0):
        for layer in self.layers:
            # velocity update: v = m*v - (dW + decay*W)
            layer.vW = momentum * layer.vW - (layer.dW + decay * layer.W)
            layer.vb = momentum * layer.vb - (layer.db + decay * layer.b)
            # weights update: w += lr * v
            layer.W += lr * layer.vW * lr  # note: multiply by lr twice is a bug; we will correct below in comment
            layer.b += lr * layer.vb * lr

    def correct_update(self, lr: float, momentum: float=0.9, decay: float=0.0):
        # Correct implementation for students to use
        for layer in self.layers:
            layer.vW = momentum * layer.vW - lr * (layer.dW + decay * layer.W)
            layer.vb = momentum * layer.vb - lr * (layer.db + decay * layer.b)
            layer.W += layer.vW
            layer.b += layer.vb

print('Layer and Model classes defined.')
```

```
Layer and Model classes defined.
```

6) Training Loop

Fill in or run the training loop. We provide a skeleton: forward, compute loss, backward, update. Use `correct_update` in practice. For MNIST quick runs use small network and fewer epochs.

```
def train_model(model: SimpleModel,
                X_train: np.ndarray, y_train: np.ndarray,
                X_val: np.ndarray, y_val: np.ndarray,
                epochs: int = 10, batch_size: int = 64,
                lr: float = 0.01, momentum: float = 0.9, decay: float = 0.0,
                loss_name: str = 'cross_entropy', verbose: bool = True):

    n = X_train.shape[0]
    history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}

    for ep in range(1, epochs+1):
        perm = np.random.permutation(n)
        X_sh = X_train[perm]
        y_sh = y_train[perm]
        epoch_loss = 0.0
```

```
            epoch_acc = 0.0
            t0 = time()
            for i in range(0, n, batch_size):
                xb = X_sh[i:i+batch_size]
                yb = y_sh[i:i+batch_size]
                # Forward
                preds = model.forward(xb)  # if final layer is softmax this is probs
                # Loss and gradient
                if loss_name == 'cross_entropy':
                    loss = cross_entropy_loss(preds, yb)
                    grad = d_cross_entropy(preds, yb)  # dL/dz when softmax final
                elif loss_name == 'l2':
                    loss = l2_loss(preds, yb)
                    grad = d_l2(preds, yb)
                elif loss_name == 'l1':
                    loss = l1_loss(preds, yb)
                    grad = d_l1(preds, yb)
                elif loss_name == 'huber':
                    loss = huber_loss(preds, yb)
                    grad = d_huber(preds, yb)
                else:
                    raise ValueError('Unknown loss')

                epoch_loss += loss * xb.shape[0]
                epoch_acc += accuracy(preds, np.argmax(yb, axis=1)) * xb.shape[0]

                # Backprop
                model.backward(grad)
                # Update weights
                model.correct_update(lr, momentum, decay)

            epoch_loss /= n
            epoch_acc /= n

            # Validation
            val_preds = model.forward(X_val)
            if loss_name == 'cross_entropy':
                val_loss = cross_entropy_loss(val_preds, y_val)
            else:
                val_loss = 0.0
            val_acc = accuracy(val_preds, np.argmax(y_val, axis=1))

            history['train_loss'].append(epoch_loss)
            history['train_acc'].append(epoch_acc)
            history['val_loss'].append(val_loss)
            history['val_acc'].append(val_acc)

            if verbose:
                print(f'Epoch {ep}/{epochs} - loss: {epoch_loss:.4f} - acc: {epoch_acc:.2f}% - val_loss: {val_loss:.4f} - val_acc: {va

        return history

    print('Training loop skeleton ready.')
```

```
Training loop skeleton ready.
```

7) Demo: MNIST Quick Setup

Load MNIST, preprocess, build a tiny model to test the pipeline. This cell is ready to run in Colab. For speed during debugging, use a subset of the data (e.g., 5000 train samples).

```
if tf_available:
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
    # flatten and normalize
    X_train = X_train.reshape(-1, 28*28).astype(np.float32) / 255.0
    X_test  = X_test.reshape(-1, 28*28).astype(np.float32) / 255.0
    # quick validation split
    X_val = X_train[-5000:]; y_val = y_train[-5000:]
    X_train = X_train[:-5000]; y_train = y_train[:-5000]
    # one hot encode
    y_train_oh = one_hot(y_train, 10)
    y_val_oh   = one_hot(y_val, 10)
    y_test_oh  = one_hot(y_test, 10)
    print('MNIST loaded:', X_train.shape, X_val.shape, X_test.shape)
```

```
    else:
        print('TensorFlow datasets not available in this environment.')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
MNIST loaded: (55000, 784) (5000, 784) (10000, 784)
```

```
# Example model creation (to run in Colab)
# Create a small model: 784 -> 128(relu) -> 10(softmax)
# NOTE: For softmax final layer, we set activation_name to 'softmax' manually.
input_dim = 784
hidden = DenseLayer(input_dim, 128, activation='relu', init='he')
output = DenseLayer(128, 10, activation='softmax', init='xavier')

model = SimpleModel([hidden, output])
print('Model ready. Run training with train_model(...)')
```

```
Model ready. Run training with train_model(...)
```

8) Experiments & Hyperparameter Tuning

Try these experiments:

- Vary learning rate: [1e-3, 1e-2, 1e-1]
- Momentum: [0.0, 0.9, 0.99]
- Weight decay: [0.0, 1e-4, 1e-3]
- Activations: relu vs tanh vs sigmoid
- Loss functions: cross-entropy vs huber

Record training curves and final test accuracy. Try CIFAR after confirming MNIST works. For CIFAR expect to tune lr and use smaller network or more epochs.

```
# ============================================================
# Imports & Utilities
# ============================================================
import numpy as np
import matplotlib.pyplot as plt
from time import time
from typing import List, Dict

try:
    from tensorflow.keras.datasets import mnist, cifar10
    tf_available = True
except:
    tf_available = False

np.random.seed(42)

# ---------- Utils ----------
def one_hot(labels: np.ndarray, num_classes: int) -> np.ndarray:
    y = np.zeros((labels.shape[0], num_classes), dtype=np.float32)
    for i, lab in enumerate(labels):
        y[i, int(lab)] = 1.0
    return y

def stable_softmax(x: np.ndarray) -> np.ndarray:
    x_max = np.max(x, axis=1, keepdims=True)
    ex = np.exp(x - x_max)
    return ex / np.sum(ex, axis=1, keepdims=True)

def accuracy(pred_probs: np.ndarray, labels: np.ndarray) -> float:
    preds = np.argmax(pred_probs, axis=1)
    return np.mean(preds == labels) * 100.0

# ============================================================
# Activations
# ============================================================
def linear(x): return x
def d_linear(out): return np.ones_like(out)

def sigmoid(x): return 1.0 / (1.0 + np.exp(-x))
def d_sigmoid(out): return out * (1.0 - out)
```

```
def tanh_act(x): return np.tanh(x)
def d_tanh(out): return 1.0 - out**2

def relu(x): return np.maximum(0, x)
def d_relu(out): return (out > 0).astype(np.float32)

def leaky_relu(x, alpha=0.01): return np.where(x >= 0, x, alpha * x)
def d_leaky_relu_out(out, alpha=0.01):
    deriv = np.ones_like(out)
    deriv[out < 0] = alpha
    return deriv

def softmax_jacobian(s: np.ndarray) -> np.ndarray:
    s = s.reshape(-1,1)
    return np.diagflat(s) - s.dot(s.T)

def d_softmax(prev_grad: np.ndarray, softmax_out: np.ndarray) -> np.ndarray:
    batch = prev_grad.shape[0]
    out = np.zeros_like(prev_grad)
    for i in range(batch):
        J = softmax_jacobian(softmax_out[i])
        out[i] = prev_grad[i].dot(J)
    return out

ACT_FNS = {
    'linear': (linear, d_linear),
    'sigmoid': (sigmoid, d_sigmoid),
    'tanh': (tanh_act, d_tanh),
    'relu': (relu, d_relu),
    'leaky_relu': (leaky_relu, d_leaky_relu_out),
}

# ============================================================
# Losses
# ============================================================
def cross_entropy_loss(probs: np.ndarray, targets: np.ndarray) -> float:
    eps = 1e-12
    probs_clipped = np.clip(probs, eps, 1. - eps)
    return -np.sum(targets * np.log(probs_clipped)) / probs.shape[0]

def d_cross_entropy(probs: np.ndarray, targets: np.ndarray) -> np.ndarray:
    batch = probs.shape[0]
    return (probs - targets) / batch

def huber_loss(pred: np.ndarray, target: np.ndarray, delta=1.0):
    diff = pred - target
    absd = np.abs(diff)
    mask = absd <= delta
    loss = np.where(mask, 0.5 * diff**2, delta * (absd - 0.5 * delta))
    return np.mean(loss)

def d_huber(pred: np.ndarray, target: np.ndarray, delta=1.0):
    diff = pred - target
    absd = np.abs(diff)
    mask = absd <= delta
    grad = np.where(mask, diff, delta * np.sign(diff))
    batch = pred.shape[0]
    return grad / batch

# ============================================================
# Layers & Model
# ============================================================
class DenseLayer:
    def __init__(self, input_dim:int, output_dim:int, activation='relu', init='xavier'):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.activation_name = activation
        self.activation, self.d_activation = ACT_FNS.get(activation, ACT_FNS['relu'])

        # init weights
        if init == 'xavier':
            limit = np.sqrt(6.0 / (input_dim + output_dim))
            self.W = np.random.uniform(-limit, limit, size=(input_dim, output_dim)).astype(np.float32)
        elif init == 'he':
            std = np.sqrt(2.0 / input_dim)
            self.W = np.random.randn(input_dim, output_dim).astype(np.float32) * std
        else:
            self.W = np.random.randn(input_dim, output_dim).astype(np.float32) * 0.01
```

```
        self.W = np.random.randn(input_dim, output_dim).astype(np.float32) * 0.01
        self.b = np.zeros((1, output_dim), dtype=np.float32)

        # gradients & velocity
        self.dW = np.zeros_like(self.W)
        self.db = np.zeros_like(self.b)
        self.vW = np.zeros_like(self.W)
        self.vb = np.zeros_like(self.b)

        self.x = None
        self.z = None
        self.a = None

    def forward(self, x: np.ndarray) -> np.ndarray:
        self.x = x
        self.z = x.dot(self.W) + self.b
        if self.activation_name == 'softmax':
            self.a = stable_softmax(self.z)
        else:
            fn = ACT_FNS[self.activation_name][0]
            self.a = fn(self.z)
        return self.a

    def backward(self, grad_a: np.ndarray) -> np.ndarray:
        if self.activation_name == 'softmax':
            grad_z = d_softmax(grad_a, self.a)
        else:
            dfn = ACT_FNS[self.activation_name][1]
            grad_z = grad_a * dfn(self.a)

        self.dW = self.x.T.dot(grad_z)
        self.db = np.sum(grad_z, axis=0, keepdims=True)
        grad_x = grad_z.dot(self.W.T)
        return grad_x

class SimpleModel:
    def __init__(self, layers: List[DenseLayer]):
        self.layers = layers

    def forward(self, x: np.ndarray) -> np.ndarray:
        out = x
        for layer in self.layers:
            out = layer.forward(out)
        return out

    def backward(self, grad_output: np.ndarray) -> None:
        grad = grad_output
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

    def correct_update(self, lr: float, momentum: float=0.9, decay: float=0.0):
        for layer in self.layers:
            layer.vW = momentum * layer.vW - lr * (layer.dW + decay * layer.W)
            layer.vb = momentum * layer.vb - lr * (layer.db + decay * layer.b)
            layer.W += layer.vW
            layer.b += layer.vb

# ============================================================
# Training Loop
# ============================================================
def train_model(model: SimpleModel,
                X_train: np.ndarray, y_train: np.ndarray,
                X_val: np.ndarray, y_val: np.ndarray,
                epochs: int = 10, batch_size: int = 64,
                lr: float = 0.01, momentum: float = 0.9, decay: float = 0.0,
                loss_name: str = 'cross_entropy', verbose: bool = True):

    n = X_train.shape[0]
    history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}

    for ep in range(1, epochs+1):
        perm = np.random.permutation(n)
        X_sh = X_train[perm]
        y_sh = y_train[perm]
        epoch_loss, epoch_acc = 0.0, 0.0
        t0 = time()
```

```
        for i in range(0, n, batch_size):
            xb = X_sh[i:i+batch_size]
            yb = y_sh[i:i+batch_size]
            preds = model.forward(xb)

            if loss_name == 'cross_entropy':
                loss = cross_entropy_loss(preds, yb)
                grad = d_cross_entropy(preds, yb)
            elif loss_name == 'huber':
                loss = huber_loss(preds, yb)
                grad = d_huber(preds, yb)
            else:
                raise ValueError('Unknown loss')

            epoch_loss += loss * xb.shape[0]
            epoch_acc += accuracy(preds, np.argmax(yb, axis=1)) * xb.shape[0]

            model.backward(grad)
            model.correct_update(lr, momentum, decay)

        epoch_loss /= n
        epoch_acc /= n
        val_preds = model.forward(X_val)
        val_loss = cross_entropy_loss(val_preds, y_val) if loss_name=='cross_entropy' else huber_loss(val_preds,y_val)
        val_acc = accuracy(val_preds, np.argmax(y_val, axis=1))

        history['train_loss'].append(epoch_loss)
        history['train_acc'].append(epoch_acc)
        history['val_loss'].append(val_loss)
        history['val_acc'].append(val_acc)

        if verbose:
            print(f'Epoch {ep}/{epochs} - loss: {epoch_loss:.4f} - acc: {epoch_acc:.2f}% - val_loss: {val_loss:.4f} - val_acc: {val

    return history

# ============================================================
# Load MNIST
# ============================================================
if tf_available:
    (X_train, y_train), (X_test, y_test) = mnist.load_data()
    X_train = X_train.reshape(-1, 28*28).astype(np.float32) / 255.0
    X_test  = X_test.reshape(-1, 28*28).astype(np.float32) / 255.0
    X_val = X_train[-5000:]; y_val = y_train[-5000:]
    X_train = X_train[:-5000]; y_train = y_train[:-5000]
    y_train_oh = one_hot(y_train, 10)
    y_val_oh   = one_hot(y_val, 10)
    y_test_oh  = one_hot(y_test, 10)
    print('MNIST loaded:', X_train.shape, X_val.shape, X_test.shape)
else:
    print("TensorFlow datasets not available.")

# ============================================================
# Hyperparameter Experiments
# ============================================================
import pandas as pd
results = []

lrs = [1e-3, 1e-2, 1e-1]
momentums = [0.0, 0.9, 0.99]
decays = [0.0, 1e-4, 1e-3]
acts = ['relu', 'tanh', 'sigmoid']
losses = ['cross_entropy', 'huber']

for lr in lrs:
    for mom in momentums:
        for wd in decays:
            for act in acts:
                for loss in losses:
                    print(f"\n>>> Training with lr={lr}, mom={mom}, wd={wd}, act={act}, loss={loss}")
                    model = SimpleModel([
                        DenseLayer(784, 128, activation=act, init='he'),
                        DenseLayer(128, 10, activation='softmax', init='xavier')
                    ])
                    history = train_model(model,
                                          X_train, y_train_oh,
                                          X_val, y_val_oh,
                                          epochs=5  # adjust to 10+ for full runs
```

```
                                    epochs=5,  # adjust to 10+ for full runs
                                    lr=lr, momentum=mom, decay=wd,
                                    loss_name=loss, verbose=False)
                    val_acc = history['val_acc'][-1]
                    test_acc = accuracy(model.forward(X_test), y_test)
                    results.append({
                        'lr': lr, 'momentum': mom, 'decay': wd,
                        'activation': act, 'loss': loss,
                        'val_acc': val_acc, 'test_acc': test_acc
                    })
                    print(f"Val acc={val_acc:.2f}% | Test acc={test_acc:.2f}%")

df = pd.DataFrame(results)
print("\n===== Summary of Experiments =====")
print(df.sort_values(by='val_acc', ascending=False).head(10))
```

```
>>> Training with lr=0.1, mom=0.99, wd=0.0, act=tanh, loss=huber
Val acc=93.68% | Test acc=92.36%

>>> Training with lr=0.1, mom=0.99, wd=0.0, act=sigmoid, loss=cross_entropy
Val acc=97.40% | Test acc=96.98%

>>> Training with lr=0.1, mom=0.99, wd=0.0, act=sigmoid, loss=huber
Val acc=97.50% | Test acc=96.98%

>>> Training with lr=0.1, mom=0.99, wd=0.0001, act=relu, loss=cross_entropy
Val acc=88.54% | Test acc=86.73%

>>> Training with lr=0.1, mom=0.99, wd=0.0001, act=relu, loss=huber
Val acc=88.86% | Test acc=86.00%

>>> Training with lr=0.1, mom=0.99, wd=0.0001, act=tanh, loss=cross_entropy
Val acc=93.18% | Test acc=91.01%

>>> Training with lr=0.1, mom=0.99, wd=0.0001, act=tanh, loss=huber
Val acc=90.64% | Test acc=88.91%

>>> Training with lr=0.1, mom=0.99, wd=0.0001, act=sigmoid, loss=cross_entropy
Val acc=96.84% | Test acc=96.10%

>>> Training with lr=0.1, mom=0.99, wd=0.0001, act=sigmoid, loss=huber
Val acc=96.72% | Test acc=95.89%

>>> Training with lr=0.1, mom=0.99, wd=0.001, act=relu, loss=cross_entropy
Val acc=85.18% | Test acc=83.34%

>>> Training with lr=0.1, mom=0.99, wd=0.001, act=relu, loss=huber
Val acc=87.32% | Test acc=85.41%

>>> Training with lr=0.1, mom=0.99, wd=0.001, act=tanh, loss=cross_entropy
Val acc=90.38% | Test acc=87.13%

>>> Training with lr=0.1, mom=0.99, wd=0.001, act=tanh, loss=huber
Val acc=89.12% | Test acc=87.17%

>>> Training with lr=0.1, mom=0.99, wd=0.001, act=sigmoid, loss=cross_entropy
Val acc=90.92% | Test acc=89.27%

>>> Training with lr=0.1, mom=0.99, wd=0.001, act=sigmoid, loss=huber
Val acc=90.80% | Test acc=88.54%

===== Summary of Experiments =====
       lr  momentum   decay activation           loss  val_acc  test_acc
90   0.01      0.99  0.0000       relu  cross_entropy    97.94     97.22
133  0.10      0.90  0.0001       relu          huber    97.90     97.26
93   0.01      0.99  0.0000       tanh          huber    97.68     96.70
97   0.01      0.99  0.0001       relu          huber    97.68     97.16
126  0.10      0.90  0.0000       relu  cross_entropy    97.68     96.93
96   0.01      0.99  0.0001       relu  cross_entropy    97.60     96.85
132  0.10      0.90  0.0001       relu  cross_entropy    97.52     97.30
92   0.01      0.99  0.0000       tanh  cross_entropy    97.50     96.80
149  0.10      0.99  0.0000    sigmoid          huber    97.50     96.98
148  0.10      0.99  0.0000    sigmoid  cross_entropy    97.40     96.98
```