HW1

September 25, 2025

1 Homework 1

This homework will cover convolutions and Canny edge detectors.

This homework was adapted from Stanford CS131

1.1 Setup

[38]:
```
# Install the necessary dependencies
# (restart your runtime session if prompted to, and then re-run this cell)

!pip install -r requirements.txt
```

```
Requirement already satisfied: numpy in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
-r requirements.txt (line 1)) (2.2.6)
Requirement already satisfied: scikit-image in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
-r requirements.txt (line 2)) (0.25.2)
Requirement already satisfied: scipy in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
-r requirements.txt (line 3)) (1.16.2)
Requirement already satisfied: matplotlib in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
-r requirements.txt (line 4)) (3.10.6)
Requirement already satisfied: networkx>=3.0 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
scikit-image->-r requirements.txt (line 2)) (3.5)
Requirement already satisfied: pillow>=10.1 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
scikit-image->-r requirements.txt (line 2)) (11.3.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
scikit-image->-r requirements.txt (line 2)) (2.37.0)
Requirement already satisfied: tifffile>=2022.8.12 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
scikit-image->-r requirements.txt (line 2)) (2025.9.20)
Requirement already satisfied: packaging>=21 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
scikit-image->-r requirements.txt (line 2)) (25.0)
Requirement already satisfied: lazy-loader>=0.4 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
scikit-image->-r requirements.txt (line 2)) (0.4)
Requirement already satisfied: contourpy>=1.0.1 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
matplotlib->-r requirements.txt (line 4)) (1.3.3)
Requirement already satisfied: cycler>=0.10 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
matplotlib->-r requirements.txt (line 4)) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
matplotlib->-r requirements.txt (line 4)) (4.59.2)
Requirement already satisfied: kiwisolver>=1.3.1 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
matplotlib->-r requirements.txt (line 4)) (1.4.9)
Requirement already satisfied: pyparsing>=2.3.1 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
matplotlib->-r requirements.txt (line 4)) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
matplotlib->-r requirements.txt (line 4)) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in
/Users/lauhityareddy/Repos/CS485_584/.conda/lib/python3.11/site-packages (from
python-dateutil>=2.7->matplotlib->-r requirements.txt (line 4)) (1.17.0)
```

[39]:

```python
# Setup
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from time import time
from skimage import io


%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 12.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

1.2 Part 1: Convolutions (20 points)

In this section, you will implement convolution: - `conv_fast`

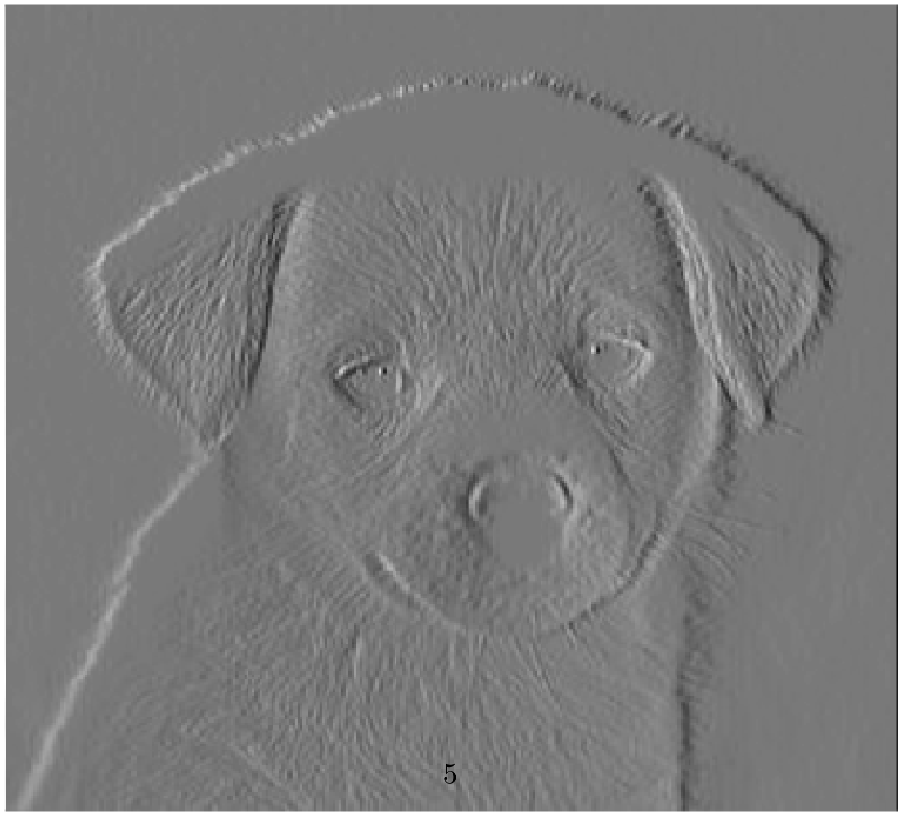
First, run the code cell below to load the image to work with.

[40]:
```
# Open image as grayscale
img = io.imread('dog.jpg', as_gray=True)

# Show image
plt.imshow(img)
plt.axis('off')
plt.show()
```

[41]:
```
# Simple convolution kernel.
# Feel free to change the kernel to see different outputs.
kernel = np.array(
[
    [1,0,-1],
    [2,0,-2],
    [1,0,-1]
])

# Plot original image
plt.subplot(2,2,1)
plt.imshow(img)
plt.title('Original')
plt.axis('off')


# Plot what you should get
solution_img = io.imread('convolved_dog.png', as_gray=True)
plt.subplot(2,2,3)
plt.imshow(solution_img)
plt.title('What you should get')
plt.axis('off')

plt.show()
```

Original

What you should get



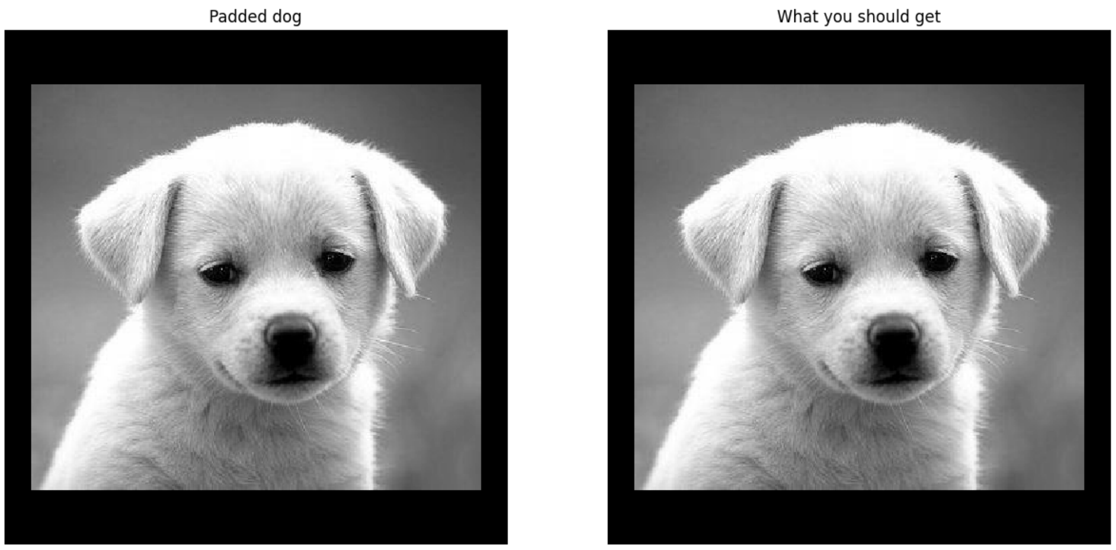
Let us implement efficient convolution using array operations in numpy. A convolution can be considered as a sliding window that computes sum of the pixel values weighted by the flipped kernel. The faster version will i) zero-pad an image, ii) flip the kernel horizontally and vertically, and iii) compute weighted sum of the neighborhood at each pixel.

First, implement the function **zero_pad** below.

[42]:
```
def zero_pad(image, pad_height, pad_width):
    """ Zero-pad an image.

    Ex: a 1x1 image [[1]] with pad_height = 1, pad_width = 2 becomes:

        [[0, 0, 0, 0, 0],
         [0, 0, 1, 0, 0],
         [0, 0, 0, 0, 0]]         of shape (3, 5)

    Args:
        image: numpy array of shape (H, W).
        pad_width: width of the zero padding (left and right padding).
        pad_height: height of the zero padding (bottom and top padding).

    Returns:
        out: numpy array of shape (H+2*pad_height, W+2*pad_width).
    """

    H, W = image.shape
    out = np.zeros((H + 2 * pad_height, W + 2 * pad_width))


    out[pad_height:pad_height+H, pad_width:pad_width+W] = image

    return out
```

[43]:
```
pad_width = 20 # width of the padding on the left and right
pad_height = 40 # height of the padding on the top and bottom


padded_img = zero_pad(img, pad_height, pad_width)

# Plot your padded dog
plt.subplot(1,2,1)
plt.imshow(padded_img)
plt.title('Padded dog')
plt.axis('off')
```

```
# Plot what you should get
solution_img = io.imread('padded_dog.jpg', as_gray=True)
plt.subplot(1,2,2)
plt.imshow(solution_img)
plt.title('What you should get')
plt.axis('off')

plt.show()
```



Next, complete the function **conv_fast** below using **zero_pad**. It would be easier to iterate over the output image (out) and map to padded image.

[44]:
```
def conv_fast(image, kernel):
    """ An efficient implementation of convolution filter.

    This function uses element-wise multiplication and np.sum()
    to efficiently compute weighted sum of neighborhood at each
    pixel.

    Hints:
        - Use the zero_pad function you implemented above
        - There should be two nested for-loops
        - You may find np.flip() and np.sum() useful

    Args:
        image: numpy array of shape (Hi, Wi).
        kernel: numpy array of shape (Hk, Wk). Dimensions will be odd.

    Returns:
        out: numpy array of shape (Hi, Wi).
    """
    Hi, Wi = image.shape
    Hk, Wk = kernel.shape
    out = np.zeros((Hi, Wi))

    padded_image = zero_pad(image, Hk//2, Wk//2)

    for i in range(Hi):
        for j in range(Wi):
            out[i, j] = np.sum(padded_image[i:i+Hk, j:j+Wk] * np.flip(kernel))

    return out
```

[45]:
```
t0 = time()
out_fast = conv_fast(img, kernel)
t1 = time()

print("conv_fast: took %f seconds." % (t1 - t0))

# Plot conv_fast output
plt.subplot(1,2,2)
plt.imshow(out_fast)
plt.title('conv_fast')
plt.axis('off')
```

```
conv_fast: took 0.281667 seconds.
```

[45]:
```
(np.float64(-0.5), np.float64(331.5), np.float64(299.5), np.float64(-0.5))
```

1.3 Part 3: Canny Edge Detector (80 points)

In this part, you are going to implement a Canny edge detector. The Canny edge detection algorithm can be broken down in to five steps: 1. Smoothing 2. Finding gradients 3. Non-maximum suppression 4. Double thresholding 5. Edge tracking by hysterisis

1.3.1 3.1 Smoothing (10 points)

We first smooth the input image by convolving it with a Gaussian kernel. The equation for a Gaussian kernel of size $(2k+1) \times (2k+1)$ is given by:

$$h_{ij} = \frac{1}{2\pi\sigma^2} \exp\Big(-\frac{(i-k)^2 + (j-k)^2}{2\sigma^2}\Big), 0 \leq i, j < 2k+1$$

Implement **`gaussian_kernel`** and run the code below.

[46]:
```
def gaussian_kernel(size, sigma):
    """ Implementation of Gaussian Kernel.

    This function follows the gaussian kernel formula,
    and creates a kernel matrix.

    Hints:
    - Use np.pi and np.exp to compute pi and exp.

    Args:
        size: int of the size of output matrix.
        sigma: float of sigma to calculate kernel.

    Returns:
        kernel: numpy array of shape (size, size).
    """

    kernel = np.zeros((size, size))

    k = size // 2
    for i in range(size):
        for j in range(size):
            kernel[i, j] = (1 / (2 * np.pi * sigma**2)) * np.exp(-((i - k)**2 +␣
 ↪(j - k)**2) / (2 * sigma**2))

    ### END YOUR CODE

    return kernel
```

[47]:
```
# Define 3x3 Gaussian kernel with std = 1
kernel = gaussian_kernel(3, 1)
kernel_test = np.array(
    [[ 0.05854983, 0.09653235, 0.05854983],
     [ 0.09653235, 0.15915494, 0.09653235],
     [ 0.05854983, 0.09653235, 0.05854983]]
)

# Test Gaussian kernel
if not np.allclose(kernel, kernel_test):
    print('Incorrect values! Please check your implementation.')
```

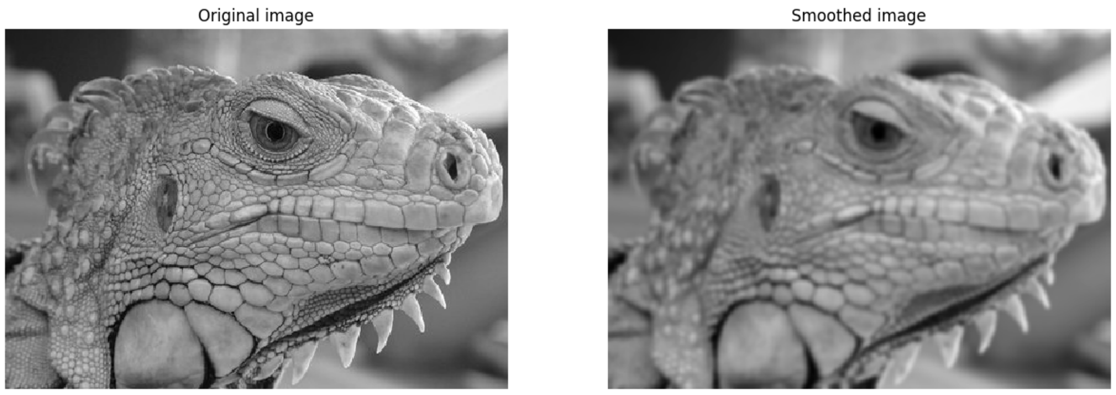
Implement **conv** and run the code below. This time, ensure that you're using **edge** padding (as opposed to zero-padding, as done in `conv_fast`).

Hint: Check out `np.pad`, and the various `modes` that it takes.

[48]:
```
def conv(image, kernel):
    """ An implementation of convolution filter.
    This function uses element-wise multiplication and np.sum()
    to efficiently compute weighted sum of neighborhood at each
    pixel.

    Args:
        image: numpy array of shape (Hi, Wi).
        kernel: numpy array of shape (Hk, Wk).

    Returns:
        out: numpy array of shape (Hi, Wi).
    """
    Hi, Wi = image.shape
    Hk, Wk = kernel.shape

    out = np.zeros((Hi, Wi),dtype=np.float32)

    # For this assignment, we will use edge values to pad the images.
    # Zero padding will make derivatives at the image boundary very big,
    # whereas we want to ignore the edges at the boundary.
    pad_height = Hk//2
    pad_width = Wk//2
    padded = np.pad(image.astype(np.float32), ((pad_height, pad_height),␣
 ↪(pad_width, pad_width)), mode='edge')

    ### YOUR CODE HERE
    flipped_kernel = np.flip(kernel)

    for i in range(Hi):
        for j in range(Wi):
            out[i, j] = np.sum(padded[i:i+Hk, j:j+Wk] * flipped_kernel)

    ### END YOUR CODE

    return out
```

[49]:
```
# Test with different kernel_size and sigma
kernel_size = 7
sigma = 1.8

# Load image
img = io.imread('iguana.png', as_gray=True)

# Define 5x5 Gaussian kernel with std = sigma
kernel = gaussian_kernel(kernel_size, sigma)

# Convolve image with kernel to achieve smoothed effect
```

```
smoothed = conv(img, kernel)

plt.subplot(1,2,1)
plt.imshow(img)
plt.title('Original image')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(smoothed)
plt.title('Smoothed image')
plt.axis('off')

plt.show()
```



1.3.2 3.2 Finding gradients (10 points)

The gradient of a 2D scalar function $I : \mathbb{R}^2 \to \mathbb{R}$ in Cartesian coordinate is defined by:

$$\nabla I(x, y) = [\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}],$$

where

$$\frac{\partial I(x,y)}{\partial x} = \lim_{\Delta x \to 0} \frac{I(x + \Delta x, y) - I(x, y)}{\Delta x} \frac{\partial I(x,y)}{\partial y} = \lim_{\Delta y \to 0} \frac{I(x, y + \Delta y) - I(x, y)}{\Delta y}.$$

In case of images, we can approximate the partial derivatives by taking differences at one pixel intervals:

$$\frac{\partial I(x,y)}{\partial x} \approx \frac{I(x+1,y) - I(x-1,y)}{2} \frac{\partial I(x,y)}{\partial y} \approx \frac{I(x,y+1) - I(x,y-1)}{2}$$

Note that the partial derivatives can be computed by convolving the image I with some appropriate kernels D_x and D_y:

$$\frac{\partial I}{\partial x} \approx I * D_x = G_x \frac{\partial I}{\partial y} \approx I * D_y = G_y$$

Find the kernels D_x and D_y and implement **`partial_x`** and **`partial_y`** using `conv` defined below.

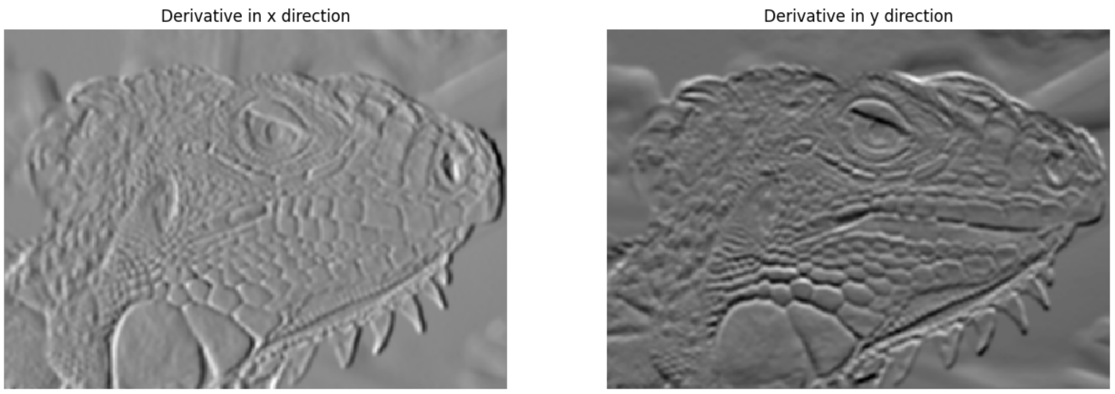
-Hint: Remeber that convolution flips the kernel.

[50]:
```
def partial_x(img):
    """ Computes partial x-derivative of input img.

    Hints:
        - You may use the conv function in defined in this file.
        - The kernel shape should be (1,3)

    Args:
        img: numpy array of shape (H, W).
    Returns:
        out: x-derivative image.
    """

    out = None

    ### YOUR CODE HERE
    # Kernel for x-derivative: [-0.5, 0, 0.5] but remember convolution flips it
    # So we use [0.5, 0, -0.5] which becomes [-0.5, 0, 0.5] after flipping
    kernel_x = np.array([[0.5, 0, -0.5]])
    out = conv(img, kernel_x)

    ### END YOUR CODE

    return out

def partial_y(img):
    """ Computes partial y-derivative of input img.

    Hints:
        - You may use the conv function in defined in this file.
        - The kernel shape should be (3,1)

    Args:
        img: numpy array of shape (H, W).
    Returns:
        out: y-derivative image.
    """

    out = None

    ### YOUR CODE HERE
    # Kernel for y-derivative: vertical version of x kernel
    # [0.5, 0, -0.5]^T which becomes [-0.5, 0, 0.5]^T after flipping
    kernel_y = np.array([[0.5], [0], [-0.5]])
    out = conv(img, kernel_y)

    ### END YOUR CODE

    return out
```

[51]:
```
# Test input
I = np.array(
    [[0, 0, 0],
     [0, 1, 0],
     [0, 0, 0]]
)

# Expected outputs
I_x_test = np.array(
    [[ 0, 0, 0],
     [ 0.5, 0, -0.5],
     [ 0, 0, 0]]
)

I_y_test = np.array(
    [[ 0, 0.5, 0],
     [ 0, 0, 0],
     [ 0, -0.5, 0]]
)

# Compute partial derivatives
I_x = partial_x(I)
I_y = partial_y(I)
print(I_x)
print(I_y)
# Test correctness of partial_x and partial_y
if not np.all(I_x == I_x_test):
    print('partial_x incorrect')

if not np.all(I_y == I_y_test):
    print('partial_y incorrect')
```

```
[[ 0.   0.   0. ]
 [ 0.5  0.  -0.5]
 [ 0.   0.   0. ]]
[[ 0.   0.5  0. ]
 [ 0.   0.   0. ]
 [ 0.  -0.5  0. ]]
```

[52]:
```python
# Compute partial derivatives of smoothed image
Gx = partial_x(smoothed)
Gy = partial_y(smoothed)

plt.subplot(1,2,1)
plt.imshow(Gx)
plt.title('Derivative in x direction')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(Gy)
plt.title('Derivative in y direction')
plt.axis('off')

plt.show()
```



Now, we can compute the magnitude and direction of gradient with the two partial derivatives:

$$G = \sqrt{G_x^2 + G_y^2} \Theta = arctan(\frac{G_y}{G_x})$$

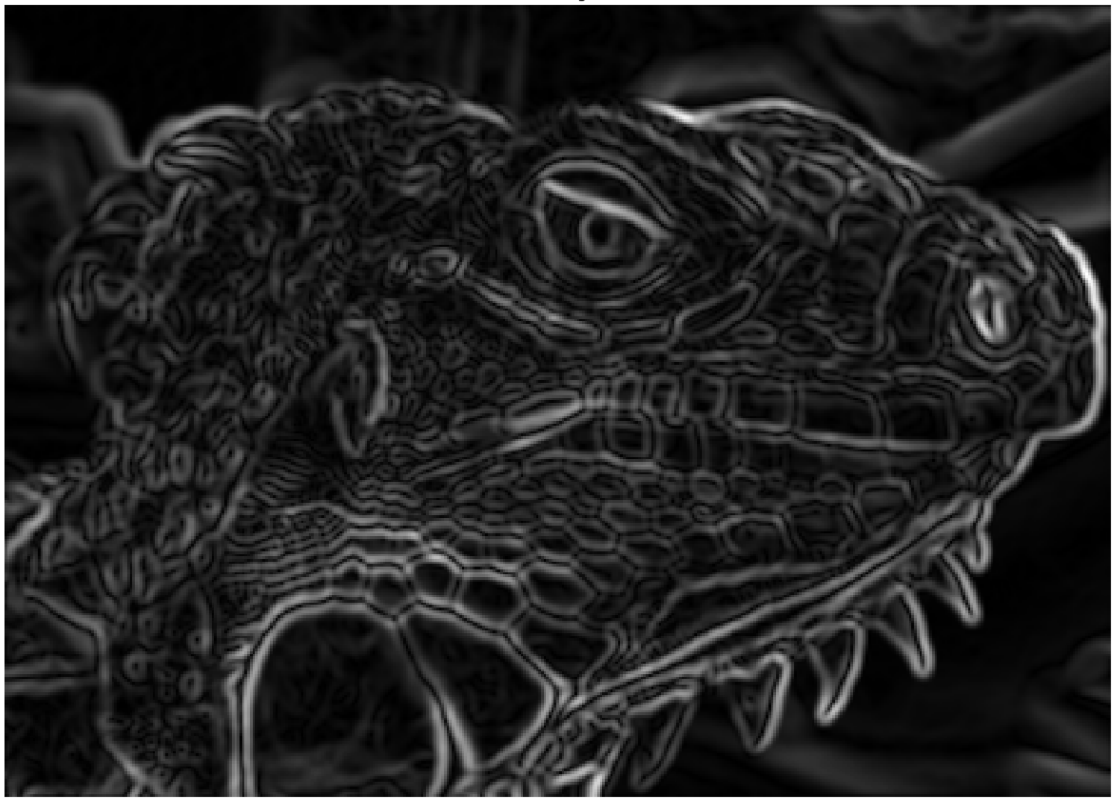
Implement **gradient** below which takes in an image and outputs G and Θ.

[53]:
```python
def gradient(img):
    """ Returns gradient magnitude and direction of input img.

    Args:
        img: Grayscale image. Numpy array of shape (H, W).

    Returns:
        G: Magnitude of gradient at each pixel in img.
            Numpy array of shape (H, W).
        theta: Direction(in degrees, 0 <= theta < 360) of gradient
```

```
            at each pixel in img. Numpy array of shape (H, W).

    Hints:
        - Use np.sqrt and np.arctan2 to calculate square root and arctan
    """
    G = np.zeros(img.shape)
    theta = np.zeros(img.shape)

    ### YOUR CODE HERE
    # Compute partial derivatives
    Gx = partial_x(img)
    Gy = partial_y(img)

    # Compute magnitude
    G = np.sqrt(Gx**2 + Gy**2)

    # Compute direction in radians, then convert to degrees
    theta_rad = np.arctan2(Gy, Gx)
    theta = np.degrees(theta_rad)

    # Convert to range [0, 360)
    theta = (theta + 360) % 360

    ### END YOUR CODE

    return G, theta
```

[54]:
```
G, theta = gradient(smoothed)

if not np.all(G >= 0):
    print('Magnitude of gradients should be non-negative.')

if not np.all((theta >= 0) * (theta < 360)):
    print('Direction of gradients should be in range 0 <= theta < 360')

plt.imshow(G)
plt.title('Gradient magnitude')
plt.axis('off')
plt.show()
```

Gradient magnitude



1.3.3 3.3 Non-maximum suppression (20 points)

You should be able to see that the edges extracted from the gradient of the smoothed image are quite thick and blurry. The purpose of this step is to convert the "blurred" edges into "sharp" edges. Basically, this is done by preserving all local maxima in the gradient image and discarding everything else. The algorithm is for each pixel (x,y) in the gradient image: 1. Round the gradient direction $\Theta[y, x]$ to the nearest 45 degrees, corresponding to the use of an 8-connected neighbourhood.

2. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions. For example, if the gradient direction is south (theta=90), compare with the pixels to the north and south.

3. If the edge strength of the current pixel is the largest; preserve the value of the edge strength. If not, suppress (i.e. remove) the value.

Implement **`non_maximum_suppression`** below.

[55]:
```
def non_maximum_suppression(G, theta):
    """ Performs non-maximum suppression.

    This function performs non-maximum suppression along the direction
    of gradient (theta) on the gradient magnitude image (G).
    Args:
        G: gradient magnitude image with shape of (H, W).
        theta: direction of gradients with shape of (H, W).

    Returns:
        out: non-maxima suppressed image.
    """
    H, W = G.shape
    out = np.zeros((H, W))

    # Round the gradient direction to the nearest 45 degrees
    theta_rounded = np.floor((theta + 22.5) / 45) * 45

    #print(G)
    ### BEGIN YOUR CODE

    for i in range(1, H-1):
        for j in range(1, W-1):
            angle = theta_rounded[i, j] % 180  # Use 180 since we have symmetry

            # Determine the two neighboring pixels to compare with
            if angle == 0:  # horizontal direction
                neighbor1 = G[i, j-1]
                neighbor2 = G[i, j+1]
            elif angle == 45:  # diagonal direction (/)
                neighbor1 = G[i-1, j+1]
                neighbor2 = G[i+1, j-1]
            elif angle == 90:  # vertical direction
                neighbor1 = G[i-1, j]
                neighbor2 = G[i+1, j]
            elif angle == 135:  # diagonal direction (\)
                neighbor1 = G[i-1, j-1]
                neighbor2 = G[i+1, j+1]
            else:
                continue

            # Check if current pixel is local maximum
            if G[i, j] >= neighbor1 and G[i, j] >= neighbor2:
                out[i, j] = G[i, j]

    ### END YOUR CODE

    return out
```

[56]:

```
# Test input
g = np.array(
    [[0.4, 0.5, 0.6],
     [0.3, 0.5, 0.7],
     [0.4, 0.5, 0.6]]
)

# Print out non-maximum suppressed output
# varying theta
for angle in range(0, 180, 45):
    #print('Thetas:', angle)
    t = np.ones((3, 3)) * angle # Initialize theta
    print(non_maximum_suppression(g, t))
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
[[0.  0.  0. ]
 [0.  0.5 0. ]
 [0.  0.  0. ]]
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

[57]:
```
nms = non_maximum_suppression(G, theta)
plt.imshow(nms)
plt.title('Non-maximum suppressed')
plt.axis('off')
plt.show()
```

Non-maximum suppressed

1.3.4 3.4 Double Thresholding (20 points)

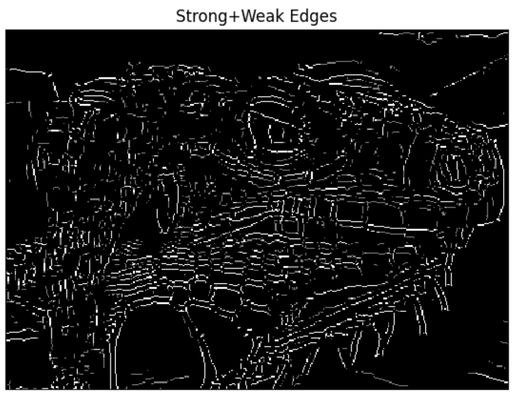
The edge-pixels remaining after the non-maximum suppression step are (still) marked with their strength pixel-by-pixel. Many of these will probably be true edges in the image, but some may be caused by noise or color variations, for instance, due to rough surfaces. The simplest way to discern between these would be to use a threshold, so that only edges stronger that a certain value would be preserved. The Canny edge detection algorithm uses double thresholding. Edge pixels stronger than the high threshold are marked as strong; edge pixels weaker than the low threshold are suppressed and edge pixels between the two thresholds are marked as weak.

Implement **double_thresholding** below.

[58]:
```python
def double_thresholding(img, high, low):
    """
    Args:
        img: numpy array of shape (H, W) representing NMS edge response.
        high: high threshold(float) for strong edges.
        low: low threshold(float) for weak edges.

    Returns:
        strong_edges: Boolean array representing strong edges.
            Strong edeges are the pixels with the values greater than
            the higher threshold.
        weak_edges: Boolean array representing weak edges.
            Weak edges are the pixels with the values smaller or equal to the
            higher threshold and greater than the lower threshold.
    """

    strong_edges = np.zeros(img.shape, dtype=bool)
    weak_edges = np.zeros(img.shape, dtype=bool)

    ### YOUR CODE HERE
    # Strong edges: values > high threshold
    strong_edges = img > high

    # Weak edges: values <= high threshold and > low threshold
    weak_edges = (img <= high) & (img > low)

    ### END YOUR CODE

    return strong_edges, weak_edges
```

[59]:
```
low_threshold = 0.02
high_threshold = 0.03

strong_edges, weak_edges = double_thresholding(nms, high_threshold,␣
 ↪low_threshold)
assert(np.sum(strong_edges & weak_edges) == 0)

edges=strong_edges * 1.0 + weak_edges * 0.5

plt.subplot(1,2,1)
plt.imshow(strong_edges)
plt.title('Strong Edges')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(edges)
plt.title('Strong+Weak Edges')
plt.axis('off')

plt.show()
```

Strong Edges



1.3.5 3.5 Edge tracking (20 points)

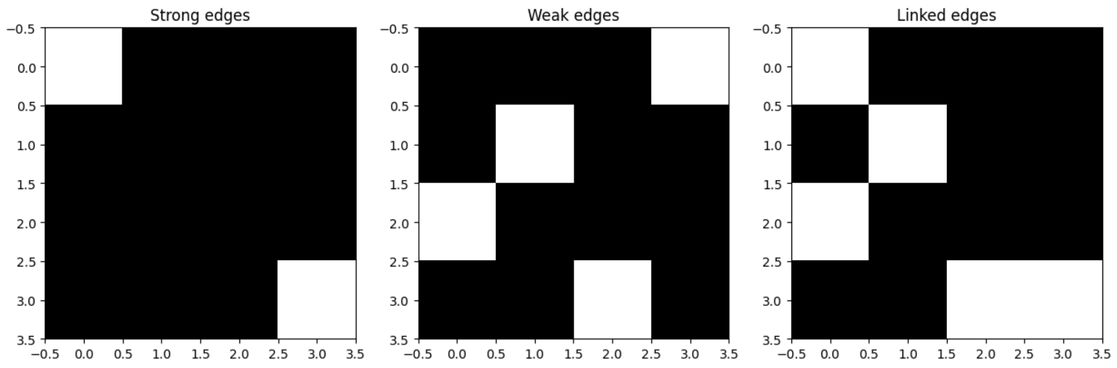
Strong edges are interpreted as "certain edges", and can immediately be included in the final edge image. Consider its neighbors iteratively then declare it an 'edge pixel' if it is connected to a 'strong edge pixel' directly or via pixels between Low and High. The logic is of course that noise and other small variations are unlikely to result in a strong edge (with proper adjustment of the threshold levels). Thus strong edges will (almost) only be due to true edges in the original image. The weak edges can either be due to true edges or noise/color variations. The latter type will probably be distributed independently of edges on the entire image, and thus only a small amount will be located adjacent to strong edges. Weak edges due to true edges are much more likely to be connected directly to strong edges.

Implement **link_edges** below.

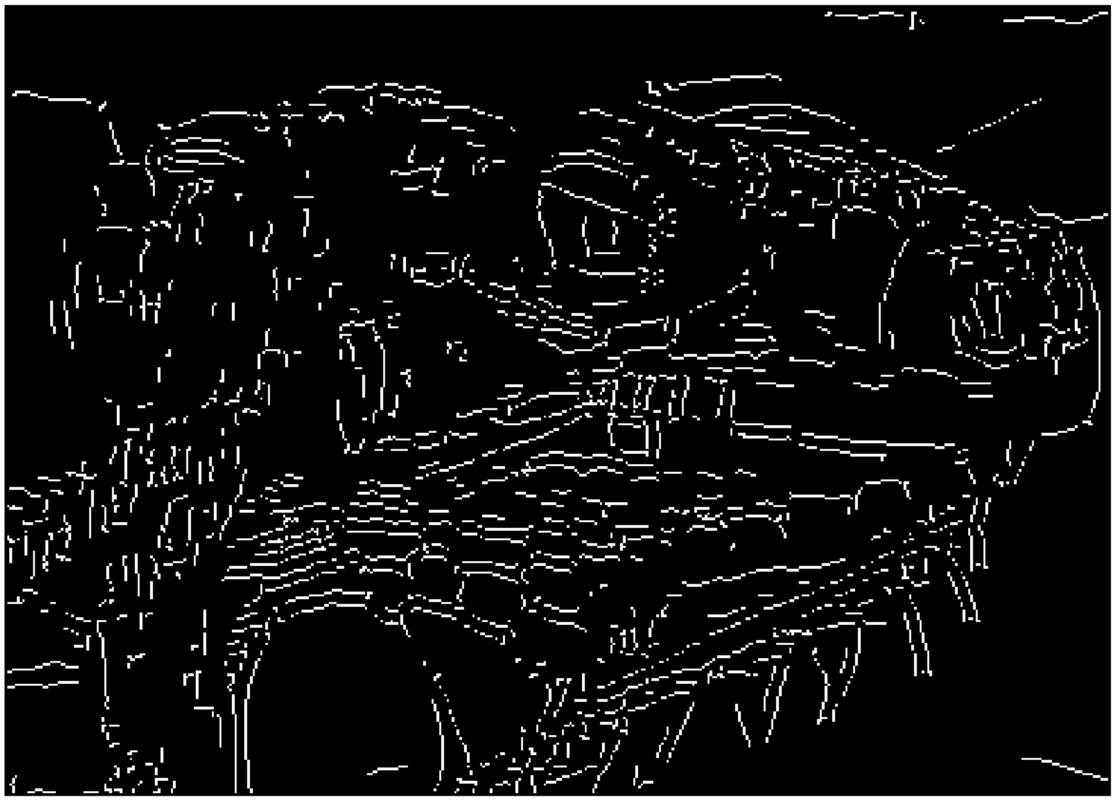
We provide the correct output and the difference between it and your result for debugging purposes. If you see white spots in the Difference image, you should check your implementation.

[60]:
```python
# a helper function for you to use:
def get_neighbors(y, x, H, W):
    """ Return indices of valid neighbors of (y, x).

    Return indices of all the valid neighbors of (y, x) in an array of
    shape (H, W). An index (i, j) of a valid neighbor should satisfy
    the following:
        1. i >= 0 and i < H
        2. j >= 0 and j < W
        3. (i, j) != (y, x)

    Args:
        y, x: location of the pixel.
        H, W: size of the image.
    Returns:
        neighbors: list of indices of neighboring pixels [(i, j)].
    """
    neighbors = []

    for i in (y-1, y, y+1):
        for j in (x-1, x, x+1):
            if i >= 0 and i < H and j >= 0 and j < W:
                if (i == y and j == x):
                    continue
                neighbors.append((i, j))

    return neighbors


def link_edges(strong_edges, weak_edges):
    """ Find weak edges connected to strong edges and link them.

    Iterate over each pixel in strong_edges and perform breadth first
    search across the connected pixels in weak_edges to link them.
    Here we consider a pixel (a, b) is connected to a pixel (c, d)
    if (a, b) is one of the eight neighboring pixels of (c, d).

    Args:
        strong_edges: binary image of shape (H, W).
        weak_edges: binary image of shape (H, W).

    Returns:
        edges: numpy boolean array of shape(H, W).
    """

    H, W = strong_edges.shape
    indices = np.stack(np.nonzero(strong_edges)).T
    edges = np.zeros((H, W), dtype=np.bool)

    # Make new instances of arguments to leave the original
    # references intact
    weak_edges = np.copy(weak_edges)
    edges = np.copy(strong_edges)

    ### YOUR CODE HERE

    # Use breadth-first search to find connected weak edges
    from collections import deque

    # For each strong edge pixel, start BFS to find connected weak edges
    for strong_pixel in indices:
        y, x = strong_pixel
        queue = deque([(y, x)])
        while queue:
            curr_y, curr_x = queue.popleft()

            # Check all 8 neighbors
            neighbors = get_neighbors(curr_y, curr_x, H, W)

            for neighbor_y, neighbor_x in neighbors:
                # If this neighbor is a weak edge and not yet marked as an edge
                if weak_edges[neighbor_y, neighbor_x] and not edges[neighbor_y,␣
↪neighbor_x]:
                    # Mark it as an edge
                    edges[neighbor_y, neighbor_x] = True
                    # Remove it from weak edges to avoid processing again
                    weak_edges[neighbor_y, neighbor_x] = False
                    # Add to queue for further exploration
                    queue.append((neighbor_y, neighbor_x))

    ### END YOUR CODE

    return edges
```

[61]:
```
test_strong = np.array(
    [[1, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 0],
     [0, 0, 0, 1]],
    dtype=np.bool
)

test_weak = np.array(
    [[0, 0, 0, 1],
     [0, 1, 0, 0],
     [1, 0, 0, 0],
     [0, 0, 1, 0]],
    dtype=np.bool
)

test_linked = link_edges(test_strong, test_weak)

plt.subplot(1, 3, 1)
plt.imshow(test_strong)
plt.title('Strong edges')

plt.subplot(1, 3, 2)
plt.imshow(test_weak)
plt.title('Weak edges')
plt.subplot(1, 3, 3)
plt.imshow(test_linked)
plt.title('Linked edges')
plt.show()
```



[62]:
```
edges = link_edges(strong_edges, weak_edges)

plt.imshow(edges)
plt.axis('off')
plt.show()
```



[ ]:
```
```