Edges

January 24, 2025

1 Edges

This notebook includes both coding and written questions. Please hand in this notebook file with all the outputs and your answers to the written questions.

This assignment covers Canny edge detector and Hough transform.

[1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cse455/assignments/assignment0/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# import os
# sys.path.append('/content/drive/MyDrive/{}'.format(FOLDERNAME))
# os.chdir('/content/drive/MyDrive/{}'.format(FOLDERNAME))
```

[2]:
```
# 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'
```

```
# for auto-reloading external modules
%load_ext autoreload
%autoreload 2
```

1.1 Part 1: Canny Edge Detector (85 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

1.1.1 1.1 Smoothing (10 points)

Implementation (5 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:

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

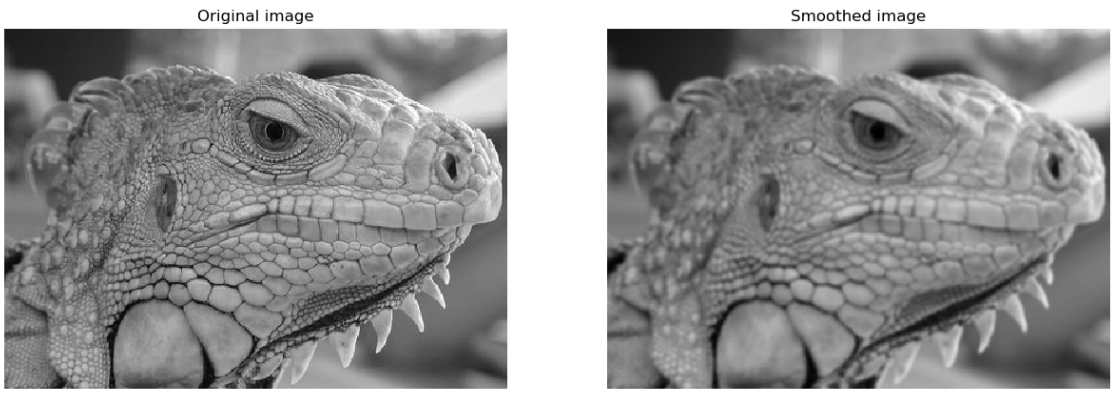
Implement **gaussian_kernel** in `edge.py` and run the code below.

[3]:
```
from edge import conv, gaussian_kernel

# 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** in `edge.py` and run the code below.

[4]:
```
# Test with different kernel_size and sigma
kernel_size = 5
sigma = 1.4

# 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()
```



Question (5 points) What is the effect of changing kernel_size and sigma?

Your Answer: It looks like increasing kernel size and sigma both make the image more blurry, and vice versa, decreasing them makes the image less blurry.

1.1.2 1.2 Finding gradients (15 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:

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

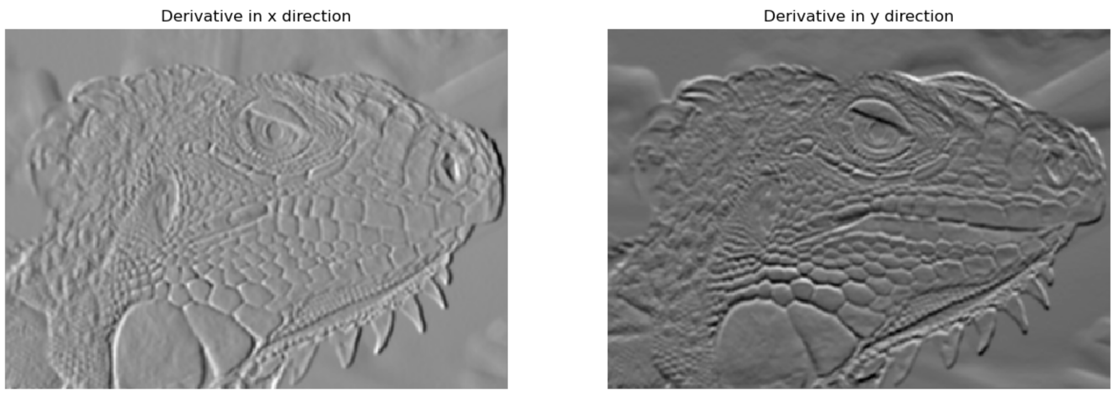
Implementation (5 points) Find the kernels D_x and D_y and implement **`partial_x`** and **`partial_y`** using `conv` defined in `edge.py`.

-Hint: Remeber that convolution flips the kernel.

[5]:
```
from edge import partial_x, partial_y

# 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)

# 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')
```

[6]:
```
# 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()
```



Question (5 points) What is the reason for performing smoothing prior to computing the gradients?

Your Answer: Write your solution in this markdown cell.

Without smoothing it is really susceptible to large changes in a single pixel value, which really shouldn't count as an edge, since an edge should be a more global thing, as we can see edges from far away but not really locally. So smoothing makes these large deviations go away, which helps reduce fake edges.

Implementation (5 points) 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`** in `edge.py` which takes in an image and outputs G and Θ.

[7]:
```
from edge import gradient

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()
```

1.1.3 1.3 Non-maximum suppression (15 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** in edge.py.

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.

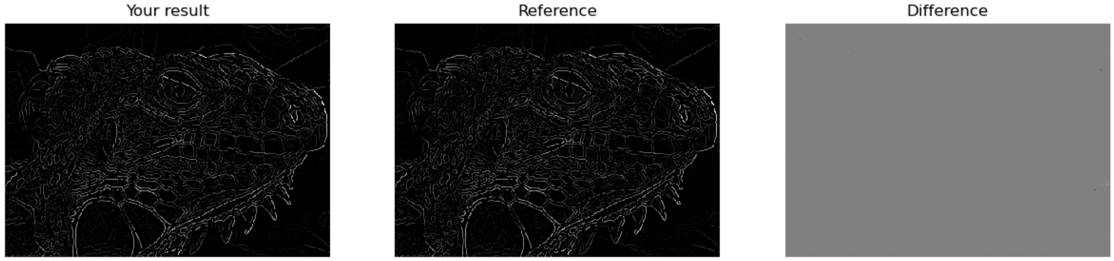
[8]:
```
from edge import non_maximum_suppression

# 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))
```

```
Thetas: 0
[[0.  0.  0.6]
 [0.  0.  0.7]
 [0.  0.  0.6]]
Thetas: 45
[[0.  0.  0.6]
 [0.  0.  0.7]
 [0.4 0.5 0.6]]
Thetas: 90
[[0.4 0.5 0. ]
 [0.  0.5 0.7]
 [0.4 0.5 0. ]]
Thetas: 135
[[0.4 0.5 0.6]
 [0.  0.  0.7]
 [0.  0.  0.6]]
```

[9]:
```
nms = non_maximum_suppression(G, theta)
plt.imshow(nms)
plt.title('Non-maximum suppressed')
plt.axis('off')
plt.show()
```

```python
plt.subplot(1, 3, 1)
plt.imshow(nms)
plt.axis('off')
plt.title('Your result')

plt.subplot(1, 3, 2)
reference = np.load('references/iguana_non_max_suppressed.npy')
plt.imshow(reference)
plt.axis('off')
plt.title('Reference')

plt.subplot(1, 3, 3)
plt.imshow(nms - reference)
plt.title('Difference')
plt.axis('off')
plt.show()

print(np.min(nms-reference), np.max(nms-reference))
np.amax(nms-reference)
```

Non-maximum suppressed




```
-0.003455310222255027 0.0034553102222550135
```

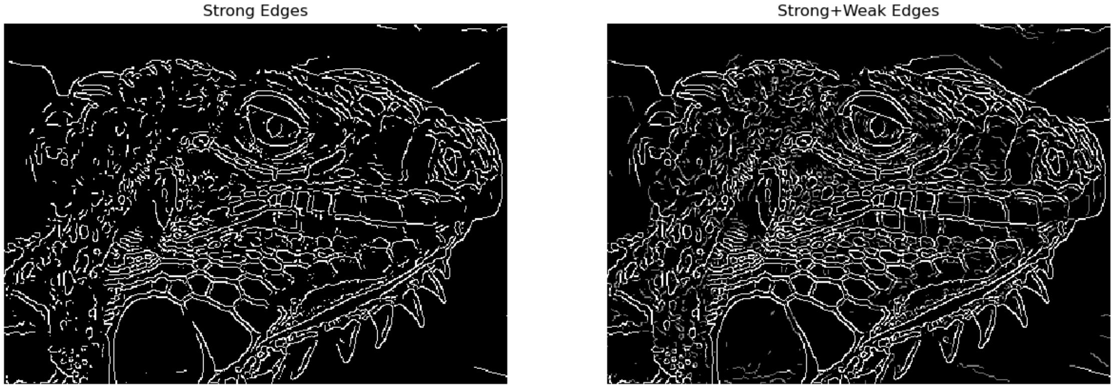
[9]: 0.0034553102222550135

1.1.4 1.4 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** in `edge.py`

[10]:
```
from edge import double_thresholding

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()
```



1.1.5 1.5 Edge tracking (15 points)

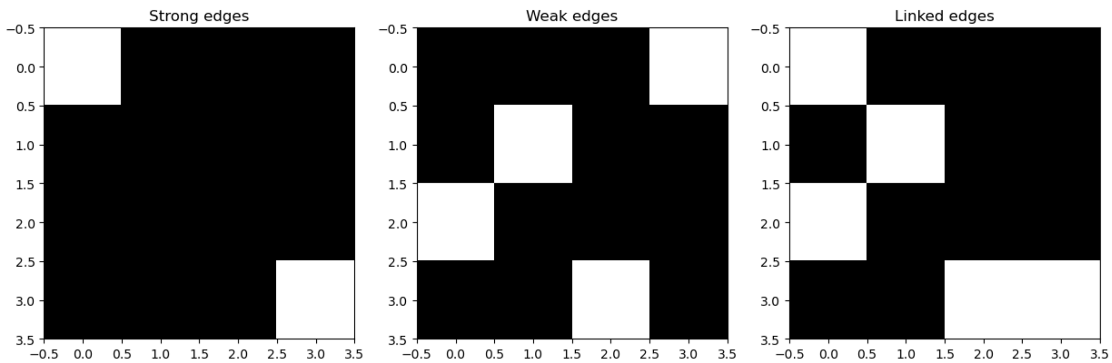
Strong edges are interpreted as “certain edges”, and can immediately be included in the final edge image. Consider its neighbors iteratively then declare it an ‘edge pixel’ if it is connected to a ‘strong edge pixel’ directly or via pixels between Low and High. The logic is of course that noise and other 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** in `edge.py`.

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.

[11]:
```
from edge import get_neighbors, link_edges

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

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



[12]:
```
edges = link_edges(strong_edges, weak_edges)

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

plt.subplot(1, 3, 1)
plt.imshow(edges)
plt.axis('off')
plt.title('Your result')

plt.subplot(1, 3, 2)
reference = np.load('references/iguana_edge_tracking.npy')
plt.imshow(reference)
plt.axis('off')
plt.title('Reference')
```

```
plt.subplot(1, 3, 3)
plt.imshow(edges ^ reference)
plt.title('Difference')
plt.axis('off')
plt.show()
```

1.1.6 1.6 Canny edge detector

Implement **canny** in `edge.py` using the functions you have implemented so far. Test edge detector with different parameters.

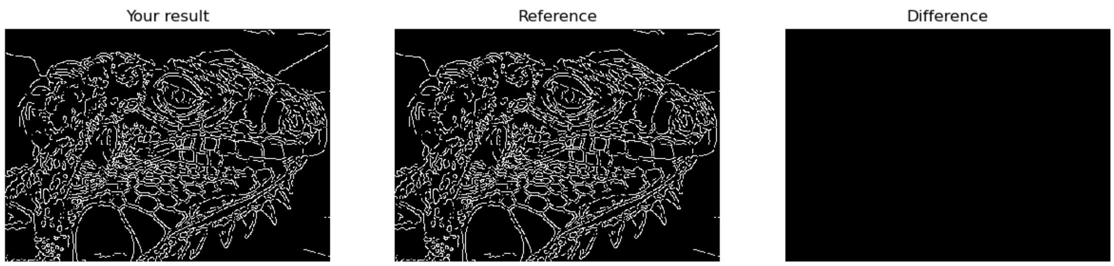
Here is an example of the output:

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.

[13]:
```python
from edge import canny

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

# Run Canny edge detector
edges = canny(img, kernel_size=5, sigma=1.4, high=0.03, low=0.02)
print (edges.shape)

plt.subplot(1, 3, 1)
plt.imshow(edges)
plt.axis('off')
plt.title('Your result')

plt.subplot(1, 3, 2)
reference = np.load('references/iguana_canny.npy')
plt.imshow(reference)
plt.axis('off')
plt.title('Reference')

plt.subplot(1, 3, 3)
```

```
plt.imshow(edges ^ reference)
plt.title('Difference')
plt.axis('off')
plt.show()
```

```
(310, 433)
```



1.1.7 Extra Credit: Optimizing Edge Detector

One way of evaluating an edge detector is to compare detected edges with manually specified ground truth edges. Here, we use precision, recall and F1 score as evaluation metrics. We provide you 40 images of objects with ground truth edge annotations. Run the code below to compute precision, recall and F1 score over the entire set of images. Then, tweak the parameters of the Canny edge detector to get as high F1 score as possible. You should be able to achieve F1 score higher than 0.31 by carefully setting the parameters.

[20]:
```
from os import listdir
from itertools import product
from tqdm import tqdm

# Define parameters to test
sigmas = [1.3, 1.7]
highs = [ 0.027, 0.025]
lows = [ 0.023, 0.018]

for sigma, high, low in product(sigmas, highs, lows):

    print("sigma={}, high={}, low={}".format(sigma, high, low))
    n_detected = 0.0
    n_gt = 0.0
    n_correct = 0.0

    for img_file in tqdm(listdir('images/objects')):
        img = io.imread('images/objects/'+img_file, as_gray=True)
        gt = io.imread('images/gt/'+img_file+'.gtf.pgm', as_gray=True)

        mask = (gt != 5) # 'don't' care region
```

```
        gt = (gt == 0) # binary image of GT edges

        edges = canny(img, kernel_size=5, sigma=sigma, high=high, low=low)
        edges = edges * mask

        n_detected += np.sum(edges)
        n_gt += np.sum(gt)
        n_correct += np.sum(edges * gt)

    p_total = n_correct / n_detected
    r_total = n_correct / n_gt
    f1 = 2 * (p_total * r_total) / (p_total + r_total)
    print('Total precision={:.4f}, Total recall={:.4f}'.format(p_total,␣
↪r_total))
    print('F1 score={:.4f}'.format(f1))
```

```
sigma=1.3, high=0.027, low=0.023

100%|      | 40/40 [01:55<00:00,  2.88s/it]

Total precision=0.0899, Total recall=0.4596
F1 score=0.1503
sigma=1.3, high=0.027, low=0.018

 15%|          | 6/40 [00:23<02:10,  3.84s/it]
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
Cell In[20], line 24
     21 mask = (gt != 5) # 'don't' care region
     22 gt = (gt == 0) # binary image of GT edges
---> 24 edges = canny(img, kernel_size=5, sigma=sigma, high=high, low=low)
     25 edges = edges * mask
     27 n_detected += np.sum(edges)

File /mnt/c/Users/Rohan Mukherjee/Documents/Homework/CSE455/assignment1/edge.py
 ↪327, in canny(img, kernel_size, sigma, high, low)
    325 smoothed = conv(img, gaussian_kernel(kernel_size, sigma))
    326 G, theta = gradient(smoothed)
--> 327 nms = non_maximum_suppression(G, theta)
    328 strong_edges, weak_edges = double_thresholding(nms, high, low)
    329 edge = link_edges(strong_edges, weak_edges)

File /mnt/c/Users/Rohan Mukherjee/Documents/Homework/CSE455/assignment1/edge.py
 ↪184, in non_maximum_suppression(G, theta)
    182 for i in range(H):
    183     for j in range(W):
--> 184         angle = np.deg2rad(theta[i,j])
    185         grad_dir_coords = (np.sin(angle), np.cos(angle))
```

```
    186         grad_dir_coords = np.round(grad_dir_coords).astype(int)

KeyboardInterrupt:
```

1.2 Part2: Lane Detection (15 points)

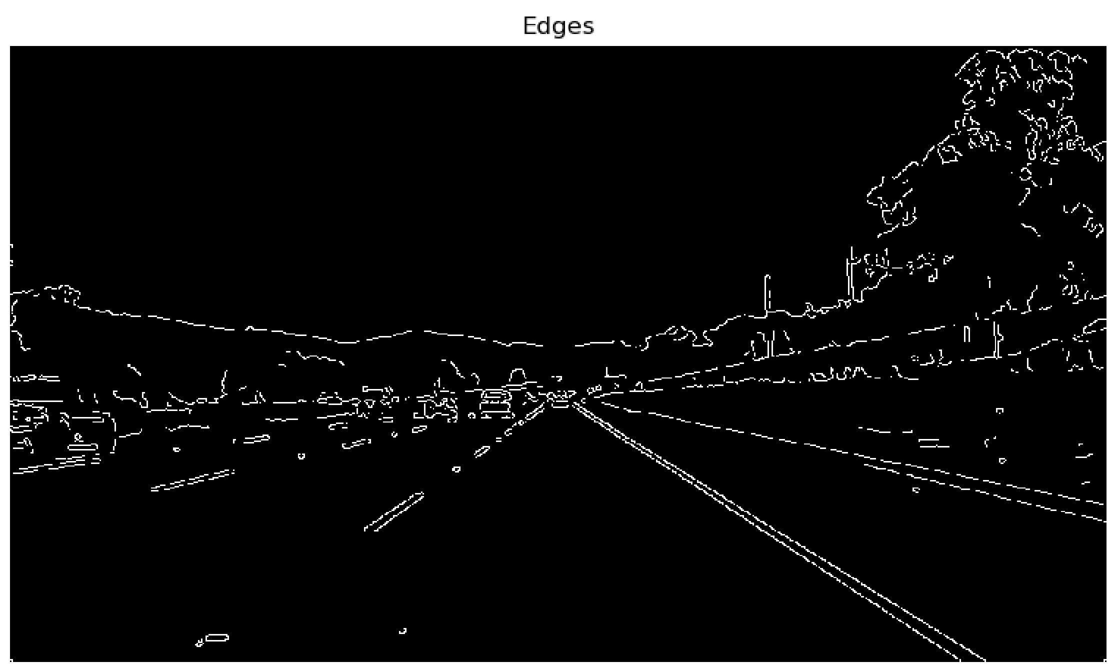
In this section we will implement a simple lane detection application using Canny edge detector and Hough transform. Here are some example images of how your final lane detector will look like.

The algorithm can broken down into the following steps: 1. Detect edges using the Canny edge detector. 2. Extract the edges in the region of interest (a triangle covering the bottom corners and the center of the image). 3. Run Hough transform to detect lanes.

1.2.1 2.1 Edge detection

Lanes on the roads are usually thin and long lines with bright colors. Our edge detection algorithm by itself should be able to find the lanes pretty well. Run the code cell below to load the example image and detect edges from the image.

[14]:
```
from edge import canny

# Load image
img = io.imread('road.jpg', as_gray=True)

# Run Canny edge detector
edges = canny(img, kernel_size=5, sigma=1.4, high=0.03, low=0.02)

plt.subplot(211)
plt.imshow(img)
plt.axis('off')
plt.title('Input Image')

plt.subplot(212)
plt.imshow(edges)
plt.axis('off')
plt.title('Edges')
plt.show()
```

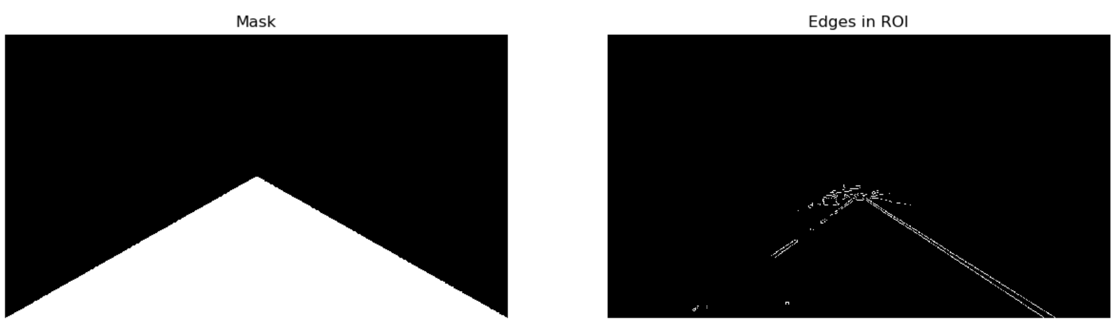


1.2.2 2.2 Extracting region of interest (ROI)

We can see that the Canny edge detector could find the edges of the lanes. However, we can also see that there are edges of other objects that we are not interested in. Given the position and orientation of the camera, we know that the lanes will be located in the lower half of the image.

The code below defines a binary mask for the ROI and extract the edges within the region.

[15]:
```
H, W = img.shape

# Generate mask for ROI (Region of Interest)
mask = np.zeros((H, W))
for i in range(H):
    for j in range(W):
        if i > (H / W) * j and i > -(H / W) * j + H:
            mask[i, j] = 1

# Extract edges in ROI
roi = edges * mask

plt.subplot(1,2,1)
plt.imshow(mask)
plt.title('Mask')
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(roi)
plt.title('Edges in ROI')
plt.axis('off')
plt.show()
```



1.2.3 2.3 Fitting lines using Hough transform (15 points)

The output from the edge detector is still a collection of connected points. However, it would be more natural to represent a lane as a line parameterized as $y = ax+b$, with a slope a and y-intercept b. We will use Hough transform to find parameterized lines that represent the detected edges.

In general, a straight line $y = ax+b$ can be represented as a point (a, b) in the parameter space. This is the parameterization we often use when introducing the Hough transform. However, this cannot represent vertical lines as the slope parameter will be unbounded. Alternatively, we parameterize a line using $\theta \in [-\pi, \pi]$ and $\rho \in \mathbb{R}$ as follows:

$$\rho = x \cdot cos\theta + y \cdot sin\theta$$

Using this parameterization, we can map every point in xy-space to a sine-like line in $\theta\rho$-space (or Hough space). We then accumulate the parameterized points in the Hough space and choose points (in Hough space) with highest accumulated values. A point in Hough space then can be transformed back into a line in xy-space.

See notes on Hough transform.

Implement **hough_transform** in edge.py.

[25]:
```
from edge import hough_transform

# Perform Hough transform on the ROI
acc, rhos, thetas = hough_transform(roi)
print('done with hough transform')

# Coordinates for right lane
xs_right = []
ys_right = []

# Coordinates for left lane
xs_left = []
ys_left = []

for i in range(20):
    idx = np.argmax(acc)
    r_idx = idx // acc.shape[1]
    t_idx = idx % acc.shape[1]
    acc[r_idx, t_idx] = 0 # Zero out the max value in accumulator

    rho = rhos[r_idx]
    theta = thetas[t_idx]

    # Transform a point in Hough space to a line in xy-space.
    a = - (np.cos(theta)/np.sin(theta)) # slope of the line
    b = (rho/np.sin(theta)) # y-intersect of the line

    # Break if both right and left lanes are detected
    if xs_right and xs_left:
        break

    if a < 0: # Left lane
        if xs_left:
            continue
        xs = xs_left
        ys = ys_left
    else: # Right Lane
```

```
        if xs_right:
            continue
        xs = xs_right
        ys = ys_right

    for x in range(img.shape[1]):
        y = a * x + b
        if y > img.shape[0] * 0.6 and y < img.shape[0]:
            xs.append(x)
            ys.append(int(round(y)))

plt.imshow(img)
plt.plot(xs_left, ys_left, linewidth=5.0)
plt.plot(xs_right, ys_right, linewidth=5.0)
plt.axis('off')
plt.show()
```

```
done with hough transform
```

Filters

January 24, 2025

1 Filters

This notebook includes both coding and written questions. Please hand in this notebook file with all the outputs and your answers to the written questions.

This assignment covers linear filters, convolution and correlation.

[1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cse455/assignments/assignment0/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# import os
# sys.path.append('/content/drive/MyDrive/{}'.format(FOLDERNAME))
# os.chdir('/content/drive/MyDrive/{}'.format(FOLDERNAME))
```

[2]:
```
# Setup
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from time import time
from skimage import io


%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading extenrnal modules
```

```
%load_ext autoreload
%autoreload 2
```

1.1 Part 1: Convolutions

1.1.1 1.1 Commutative Property (5 points)

Recall that the convolution of an image $f : \mathbb{R}^2 \to \mathbb{R}$ and a kernel $h : \mathbb{R}^2 \to \mathbb{R}$ is defined as follows:

$$(f * h)[m, n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i, j] \cdot h[m - i, n - j]$$

Or equivalently,

$$\begin{aligned}(f * h)[m, n] &= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} h[i, j] \cdot f[m - i, n - j] &(1)\\ &= (h * f)[m, n] &(2)\end{aligned}$$

Show that this is true (i.e. prove that the convolution operator is commutative: $f * h = h * f$).

Your Answer: *Write your solution in this markdown cell. Please write your equations in [LaTex equations](#).*

Consider change of variables $(i, j) \mapsto (m - i, n - j)$. This is a bijection from $\mathbb{Z}^2 \to \mathbb{Z}^2$ as it has an inverse with integer coefficients, namely, itself. Then, using this change of variables we can see that:

$$(f * h)[m, n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i, j] \cdot h[m - i, n - j]$$

$$= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[m - i, n - j] h[m - (m - i), n - (n - j)] = (h * f)[m, n]$$

1.1.2 1.2 Shift Invariance (5 points)

Let f be a function $\mathbb{R}^2 \to \mathbb{R}$. Consider a system $f \xrightarrow{s} g$, where $g = (f * h)$ with some kernel $h : \mathbb{R}^2 \to \mathbb{R}$. Also consider functions $f'[m, n] = f[m - m_0, n - n_0]$ and $g'[m, n] = g[m - m_0, n - n_0]$.

Show that S defined by any kernel h is a shift invariant system by showing that $g' = (f' * h)$.

Your Answer: *Write your solution in this markdown cell. Please write your equations in [LaTex equations](#).*

This really cleared up a confusion of mine that I didn't feel was clarified in class, which is that shift invariance means that the system commutes with the shift operator. For example, rotation doesn't, since if you rotate then shift, you get a different answer than shifting and then rotating (this rotates about a different point).

For this, we write:

$$g'[m, n] = g[m - m_0, n - n_0] = \sum_{i,j=-\infty}^{\infty} f[m - m_0 - i, n - n_0 - j] \cdot h[i, j]$$

$$= \sum_{i,j} f'[m-i, n-j]h[i,j] = (f' * h)[m,n]$$

1.1.3 1.3 Linearity (10 points)

Recall that a system S is considered a linear system if and only if it satisfies the superposition property. In mathematical terms, a (function) S is a linear invariant system iff it satisfies:

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \alpha S\{f_1[n,m]\} + \beta S\{f_2[n,m]\}$$

Let f_1 and f_2 be functions $\mathbb{R}^2 \to \mathbb{R}$. Consider a system $f \xrightarrow{s} g$, where $g = (f * h)$ with some kernel $h : \mathbb{R}^2 \to \mathbb{R}$.

Prove that S defined by any kernel h is linear by showing that the superposition property holds.

Your Answer: *Write your solution in this markdown cell. Please write your equations in [LaTex equations](#).*

$$S\{\alpha f_1[n,m] + \beta f_2[n,m]\} = \sum_{i,j=-\infty}^{\infty} (\alpha f_1[n-i, m-j] + \beta f_2[n-i, m-j]) \cdot h[i,j]$$

$$= \alpha \sum_{i,j} f_1[n-i, m-j]h[i,j] + \beta \sum_{i,j} f_2[n-i, m-j]h[i,j]$$

$$= \alpha S\{f_1[n,m]\} + \beta S\{f_2[n,m]\}$$

1.1.4 1.4 Implementation (30 points)

In this section, you will implement two versions of convolution: - `conv_nested` - `conv_fast`

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

[3]:
```python
# Open image as grayscale
img = io.imread('dog.jpg', as_gray=True)

# Show image
plt.imshow(img)
plt.axis('off')
plt.title("Isn't he cute?")
plt.show()
```

Isn't he cute?

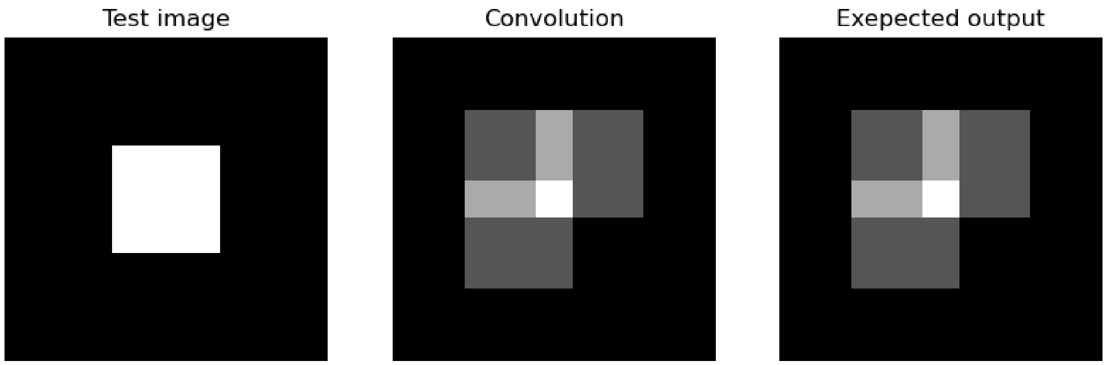
Now, implement the function **`conv_nested`** in **`filters.py`**. This is a naive implementation of convolution which uses 4 nested for-loops. It takes an image f and a kernel h as inputs and outputs the convolved image $(f * h)$ that has the same shape as the input image. This implementation should take a few seconds to run.

- *Hint: It may be easier to implement* $(h * f)$

We'll first test your `conv_nested` function on a simple input.

[4]:
```
from filters import conv_nested

# Simple convolution kernel.
kernel = np.array(
[
    [1,0,1],
    [0,0,0],
```

```
    [1,0,0]
])

# Create a test image: a white square in the middle
test_img = np.zeros((9, 9))
test_img[3:6, 3:6] = 1

# Run your conv_nested function on the test image
test_output = conv_nested(test_img, kernel)

# Build the expected output
expected_output = np.zeros((9, 9))
expected_output[2:7, 2:7] = 1
expected_output[5:, 5:] = 0
expected_output[4, 2:5] = 2
expected_output[2:5, 4] = 2
expected_output[4, 4] = 3

# Plot the test image
plt.subplot(1,3,1)
plt.imshow(test_img)
plt.title('Test image')
plt.axis('off')

# Plot your convolved image
plt.subplot(1,3,2)
plt.imshow(test_output)
plt.title('Convolution')
plt.axis('off')

# Plot the exepected output
plt.subplot(1,3,3)
plt.imshow(expected_output)
plt.title('Exepected output')
plt.axis('off')
plt.show()

# Test if the output matches expected output
assert np.max(test_output - expected_output) < 1e-10, "Your solution is not␣
 ↪correct."
```



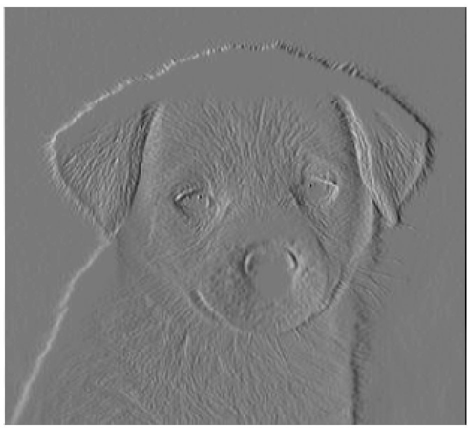
Now let's test your `conv_nested` function on a real image.

[5]:
```
from filters import conv_nested

# 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]
])

out = conv_nested(img, kernel)

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

# Plot your convolved image
plt.subplot(2,2,3)
plt.imshow(out)
plt.title('Convolution')
plt.axis('off')

# Plot what you should get
solution_img = io.imread('convolved_dog.png', as_gray=True)
plt.subplot(2,2,4)
plt.imshow(solution_img)
plt.title('What you should get')
plt.axis('off')
```

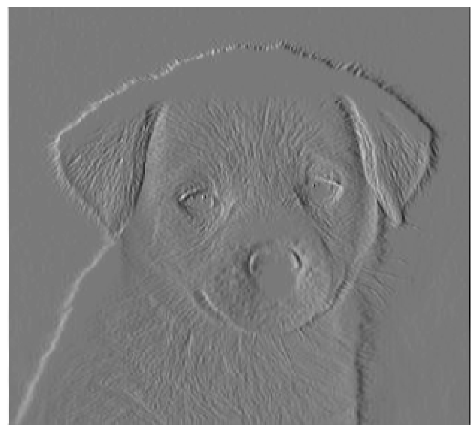
```
plt.show()
```

Original

Convolution



What you should get

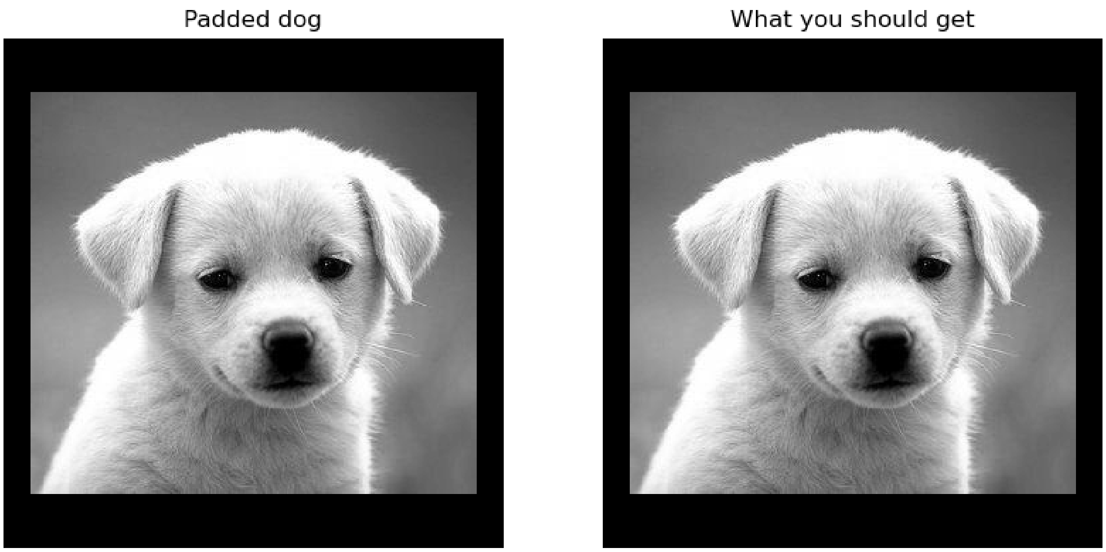


Let us implement a more efficient version of convolution using array operations in numpy. As shown in the lecture, a convolution can be considered as a sliding window that computes sum of the pixel values weighted by the flipped kernel. The faster version will i) zero-pad an image, ii) flip the kernel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

First, implement the function **zero_pad** in **filters.py**.

[6]:
```
from filters import zero_pad

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()
```

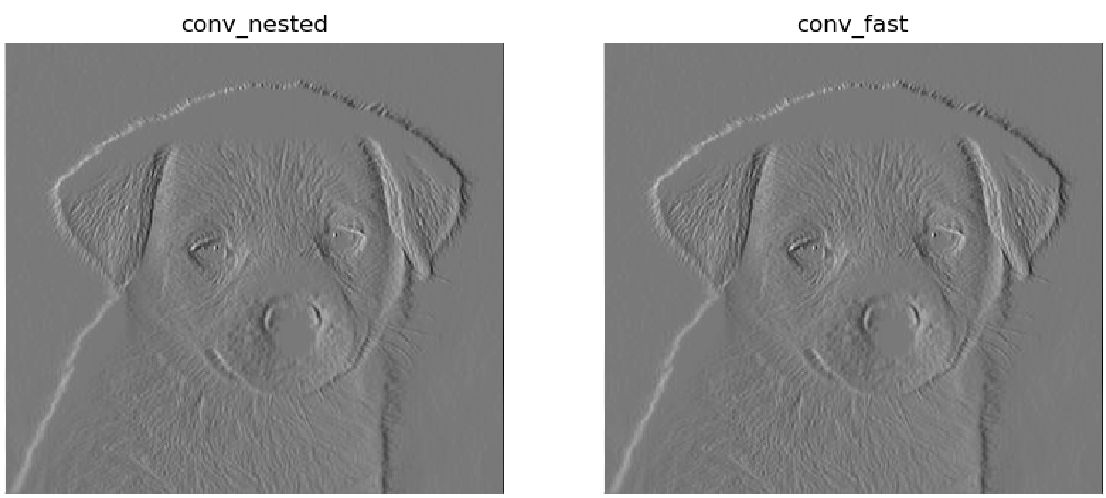


Next, complete the function **conv_fast** in **filters.py** using zero_pad. Run the code below to compare the outputs by the two implementations. conv_fast should run noticeably faster than conv_nested.

[7]:
```
from filters import conv_fast

t0 = time()
out_fast = conv_fast(img, kernel)
t1 = time()
out_nested = conv_nested(img, kernel)
t2 = time()

# Compare the running time of the two implementations
print("conv_nested: took %f seconds." % (t2 - t1))
```

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

# Plot conv_nested output
plt.subplot(1,2,1)
plt.imshow(out_nested)
plt.title('conv_nested')
plt.axis('off')

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

plt.show()

# Make sure that the two outputs are the same
if not (np.max(out_fast - out_nested) < 1e-10):
    print("Different outputs! Check your implementation.")
```

```
conv_nested: took 0.303510 seconds.
conv_fast: took 0.249335 seconds.
```



1.2 Part 2: Cross-correlation

Cross-correlation of an image f with a template g is defined as follows:

$$(g ** f)[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} g[i,j] \cdot f[m+i, n+j]$$

1.2.1 2.1 Template Matching with Cross-correlation (12 points)

Suppose that you are a clerk at a grocery store. One of your responsibilites is to check the shelves periodically and stock them up whenever there are sold-out items. You got tired of this laborious task and decided to build a computer vision system that keeps track of the items on the shelf.

Luckily, you have learned in the course that cross-correlation can be used for template matching: a template g is multiplied with regions of a larger image f to measure how similar each region is to the template.

The template of a product (`template.jpg`) and the image of shelf (`shelf.jpg`) is provided. We will use cross-correlation to find the product in the shelf.

Implement **`cross_correlation`** function in **`filters.py`** and run the code below.

*- Hint: you may use the **`conv_fast`** function you implemented in the previous question.*

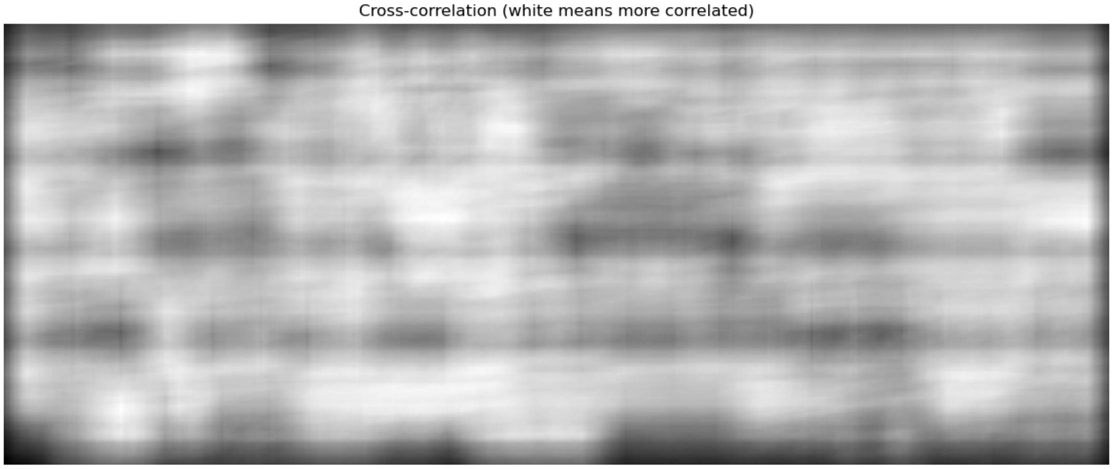
[9]:
```python
from filters import cross_correlation

# Load template and image in grayscale
img = io.imread('shelf.jpg')
img_gray = io.imread('shelf.jpg', as_gray=True)
temp = io.imread('template.jpg')
temp_gray = io.imread('template.jpg', as_gray=True)

# Perform cross-correlation between the image and the template
out = cross_correlation(img_gray, temp_gray)

# output sizes
print("Image size:", img_gray.shape)
print("Output size:", out.shape)

# Find the location with maximum similarity
y,x = (np.unravel_index(out.argmax(), out.shape))

# Display product template
plt.figure(figsize=(25,20))
plt.subplot(3, 1, 1)
plt.imshow(temp)
plt.title('Template')
plt.axis('off')

# Display cross-correlation output
plt.subplot(3, 1, 2)
plt.imshow(out)
plt.title('Cross-correlation (white means more correlated)')
plt.axis('off')

# Display image
plt.subplot(3, 1, 3)
```

```python
plt.imshow(img)
plt.title('Result (blue marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'bx', ms=40, mew=10)
plt.show()
```

```
Image size: (400, 1000)
Output size: (400, 1000)
```

Template

Cross-correlation (white means more correlated)



Result (blue marker on the detected location)

Interpretation How does the output of cross-correlation filter look? Explain what problems there might be with using a raw template as a filter.

Your Answer: *Write your solution in this markdown cell.* What it seems like to me is that red pixels are often darker than everything else, even though they should be the most correlated. I think the issue here is that if I am comparing a dark template with a light spot of the image, just due to the multiplication this will yield a higher value than comparing the dark template with itself, which makes the "correlation" go up when it really shouldn't. I believe this is why it puts the X on the lightest spot of the image.

1.2.2 2.2 Zero-mean cross-correlation (6 points)

A solution to this problem is to subtract the mean value of the template so that it has zero mean.

Implement **`zero_mean_cross_correlation`** function in **`filters.py`** and run the code below.

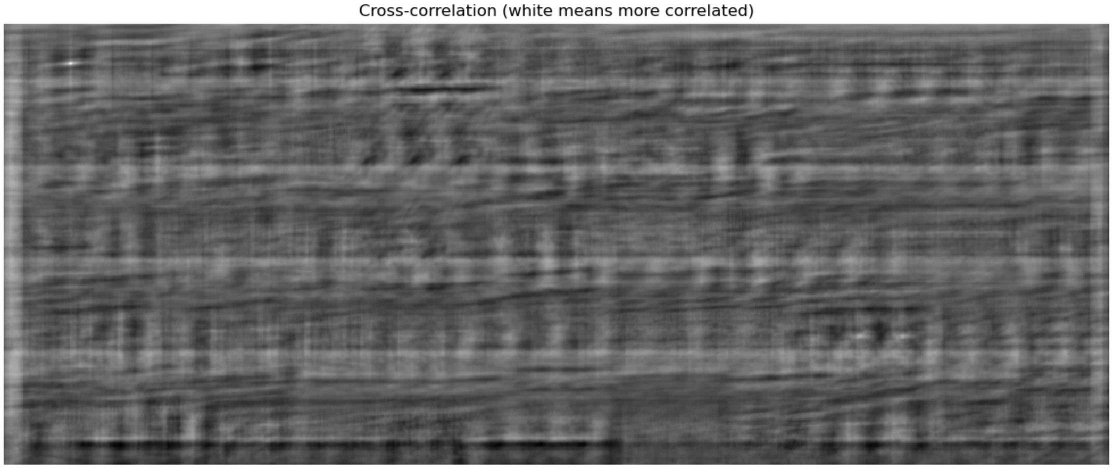
If your implementation is correct, you should see the blue cross centered over the correct cereal box.

[34]:
```python
from filters import zero_mean_cross_correlation

# Perform cross-correlation between the image and the template
out = zero_mean_cross_correlation(img_gray, temp_gray)

# Find the location with maximum similarity
y,x = np.unravel_index(out.argmax(), out.shape)

# Display product template
plt.figure(figsize=(30,20))
plt.subplot(3, 1, 1)
plt.imshow(temp)
plt.title('Template')
plt.axis('off')

# Display cross-correlation output
plt.subplot(3, 1, 2)
plt.imshow(out)
plt.title('Cross-correlation (white means more correlated)')
plt.axis('off')

# Display image
plt.subplot(3, 1, 3)
plt.imshow(img)
plt.title('Result (blue marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'bx', ms=40, mew=10)
```

```
plt.show()
```

Template

Cross-correlation (white means more correlated)



Result (blue marker on the detected location)

You can also determine whether the product is present with appropriate scaling and thresholding.

[35]:
```python
def check_product_on_shelf(shelf, product):
    out = zero_mean_cross_correlation(shelf, product)

    # Scale output by the size of the template
    out = out / float(product.shape[0]*product.shape[1])

    # Threshold output (this is arbitrary, you would need to tune the threshold for a real application)
    out = out > 0.025

    if np.sum(out) > 0:
        print('The product is on the shelf')
    else:
        print('The product is not on the shelf')

# Load image of the shelf without the product
img2 = io.imread('shelf_soldout.jpg')
img2_gray = io.imread('shelf_soldout.jpg', as_gray=True)

plt.imshow(img)
plt.axis('off')
plt.show()
check_product_on_shelf(img_gray, temp_gray)

plt.imshow(img2)
plt.axis('off')
plt.show()
check_product_on_shelf(img2_gray, temp_gray)
```

```
The product is on the shelf
```

```
The product is not on the shelf
```

1.2.3 2.3 Normalized Cross-correlation (12 points)

One day the light near the shelf goes out and the product tracker starts to malfunction. The `zero_mean_cross_correlation` is not robust to change in lighting condition. The code below demonstrates this.

[36]:
```python
from filters import normalized_cross_correlation

# Load image
img = io.imread('shelf_dark.jpg')
img_gray = io.imread('shelf_dark.jpg', as_gray=True)

# Perform cross-correlation between the image and the template
out = zero_mean_cross_correlation(img_gray, temp_gray)

# Find the location with maximum similarity
y, x = np.unravel_index(out.argmax(), out.shape)

# Display image
plt.imshow(img)
plt.title('Result (red marker on the detected location)')
plt.axis('off')

# Draw marker at detected location
plt.plot(x, y, 'rx', ms=25, mew=5)
plt.show()
```

A solution is to normalize the pixels of the image and template at every step before comparing them. This is called **normalized cross-correlation**.

The mathematical definition for normalized cross-correlation of f and template g is:

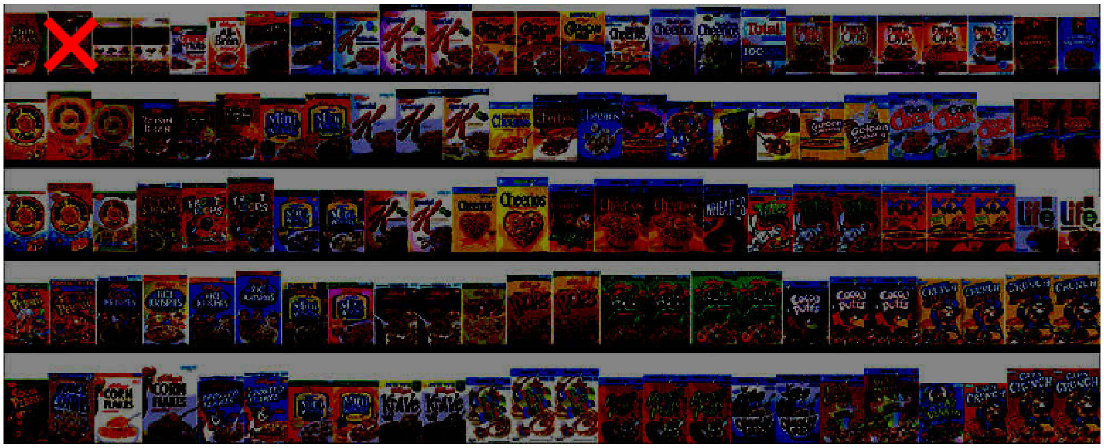
$$(g \star f)[m,n] = \sum_{i,j} \frac{g[i,j] - \overline{g}}{\sigma_g} \cdot \frac{f[m+i,n+j] - \overline{f_{m,n}}}{\sigma_{f_{m,n}}}$$

where: - $f_{m,n}$ is the patch image at position (m,n) - $\overline{f_{m,n}}$ is the mean of the patch image $f_{m,n}$ - $\sigma_{f_{m,n}}$ is the standard deviation of the patch image $f_{m,n}$ - $\overline{g}$ is the mean of the template g - σ_g is the standard deviation of the template g

Implement **`normalized_cross_correlation`** function in **`filters.py`** and run the code below.

[37]:
```python
from filters import normalized_cross_correlation

# Perform normalized cross-correlation between the image and the template
out = normalized_cross_correlation(img_gray, temp_gray)

# Find the location with maximum similarity
y, x = np.unravel_index(out.argmax(), out.shape)

# Display image
plt.imshow(img)
plt.title('Result (red marker on the detected location)')
plt.axis('off')

# Draw marker at detcted location
plt.plot(x, y, 'rx', ms=25, mew=5)
plt.show()
```

Result (red marker on the detected location)



1.3 Part 3: Separable Filters

1.3.1 3.1 Theory (10 points)

Consider an $M_1 \times N_1$ image I and an $M_2 \times N_2$ filter F. A filter F is **separable** if it can be written as a product of two 1D filters: $F = F_1 F_2$.

For example,

$$F = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

can be written as a matrix product of

$$F_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, F_2 = \begin{bmatrix} 1 & -1 \end{bmatrix}$$

Therefore F is a separable filter.

Prove that for any separable filter $F = F_1 F_2$,

$$I * F = (I * F_1) * F_2$$

where $*$ is the convolution operation.

Your Answer: *Write your solution in this markdown cell. Please write your equations in LaTex equations.*

We can write the convolution of I and F_1 as:

$$(I * F_1)[m, n] = \sum_{i=1}^{M_2} I[m - i, n] F_1[i]$$

Then, as $F[i, j] = F_1[i] F_2[j]$ by definition,

$$(I * F_1) * F_2[m, n] = \sum_{j=1}^{N_2} (I * F_1)[m, n - j] F_2[j] = \sum_{j=1}^{N_2} \sum_{i=1}^{M_2} I[m - i, n - j] F_1[i] F_2[j] = (I * F)[m, n]$$

1.3.2 3.2 Complexity comparison (10 points)

Consider an $M_1 \times N_1$ image I and an $M_2 \times N_2$ filter F that is separable (i.e. $F = F_1 F_2$).

(i) How many multiplication operations do you need to do a direct 2D convolution (i.e. $I * F$)?
(ii) How many multiplication operations do you need to do 1D convolutions on rows and columns (i.e. $(I * F_1) * F_2$)?
(iii) Use Big-O notation, written with respet to the dimensions M_1, N_1, M_2, and N_2, to argue which one is more efficient in general: direct 2D convolution or two successive 1D convolutions?

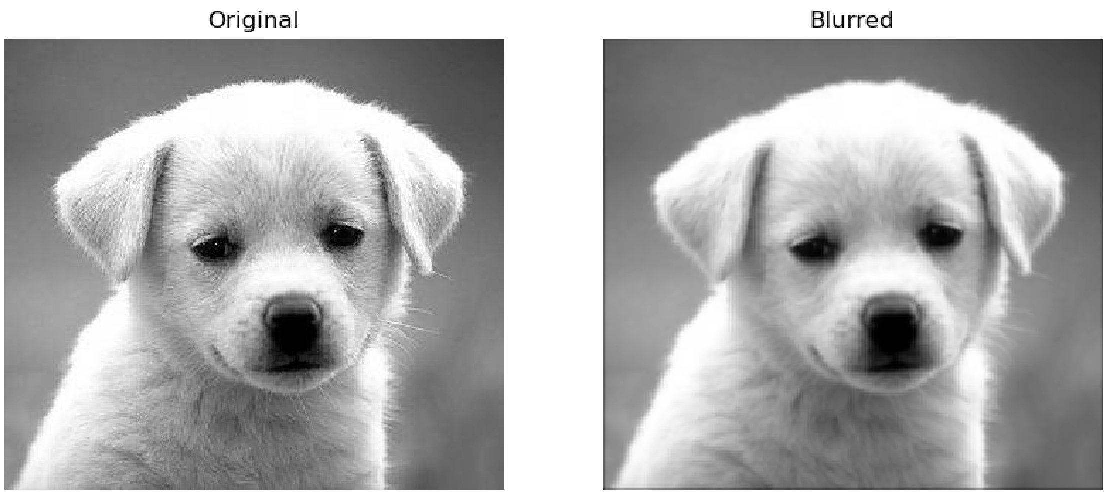
Your Answer: *Write your solution in this markdown cell. Please write your equations in [LaTex equations](#).* To do a direct 2d convolution, for every (m, n) index in the image, of which there are $M_1 N_1$, we have to do a sum with $M_2 N_2$ terms each of which has 1 multiplication. This gives $M_1 N_1 M_2 N_2$ multiplications.

On the other hand, for the first 1d convolution, for each index (m, n), we have to do M_2 multiplications. Once that is done, for each index we have to do N_2 multiplications per sum. This gives $M_1 N_1 (M_2 + N_2)$ multiplications, which is indeed much faster.

Now, we will empirically compare the running time of a separable 2D convolution and its equivalent two 1D convolutions. The Gaussian kernel, widely used for blurring images, is one example of a separable filter. Run the code below to see its effect.

[38]:
```
# Load image
img = io.imread('dog.jpg', as_gray=True)

# 5x5 Gaussian blur
kernel = np.array([
    [1,4,6,4,1],
    [4,16,24,16,4],
    [6,24,36,24,6],
    [4,16,24,16,4],
    [1,4,6,4,1]
])

t0 = time()
out = conv_nested(img, kernel)
t1 = time()
t_normal = t1 - t0

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

# Plot convolved image
plt.subplot(1,2,2)
plt.imshow(out)
```

```
plt.title('Blurred')
plt.axis('off')

plt.show()
```

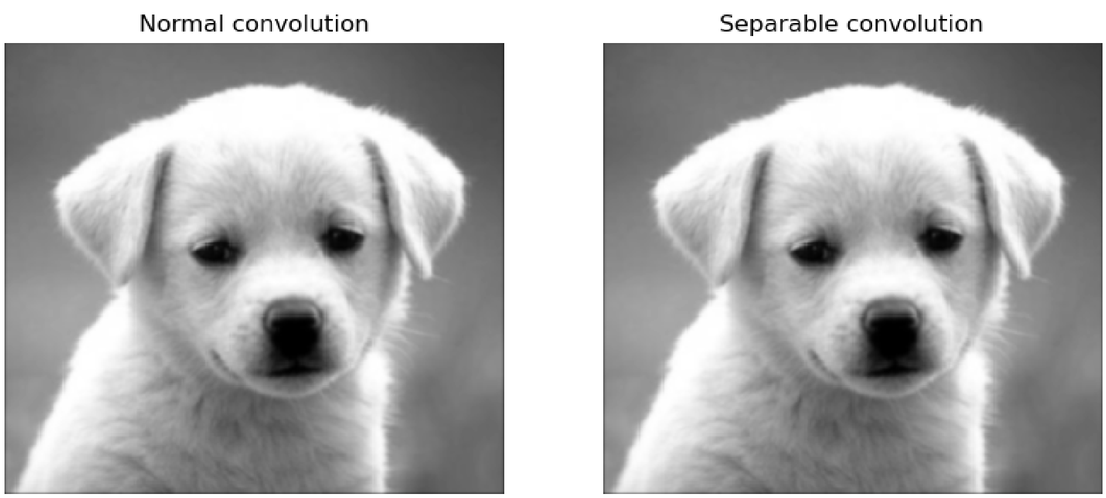


In the below code cell, define the two 1D arrays (`k1` and `k2`) whose product is equal to the Gaussian kernel.

[40]:
```
# The kernel can be written as outer product of two 1D filters
k1 = None  # shape (5, 1)
k2 = None # shape (1, 5)

### YOUR CODE HERE
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

k1 = np.array([1, 4, 6, 4, 1]).reshape(-1,1)
k2 = k1.T

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
### END YOUR CODE

# Check if kernel is product of k1 and k2
if not  np.all(k1 * k2 == kernel):
    print('k1 * k2 is not equal to kernel')

assert k1.shape == (5, 1), "k1 should have shape (5, 1)"
assert k2.shape == (1, 5), "k2 should have shape (1, 5)"
```

We now apply the two versions of convolution to the same image, and compare their running time. Note that the outputs of the two convolutions must be the same.

[41]:
```python
# Perform two convolutions using k1 and k2
t0 = time()
out_separable = conv_nested(img, k1)
out_separable = conv_nested(out_separable, k2)
t1 = time()
t_separable = t1 - t0

# Plot normal convolution image
plt.subplot(1,2,1)
plt.imshow(out)
plt.title('Normal convolution')
plt.axis('off')

# Plot separable convolution image
plt.subplot(1,2,2)
plt.imshow(out_separable)
plt.title('Separable convolution')
plt.axis('off')

plt.show()

print("Normal convolution: took %f seconds." % (t_normal))
print("Separable convolution: took %f seconds." % (t_separable))
```




```
Normal convolution: took 0.945580 seconds.
Separable convolution: took 0.347029 seconds.
```

[42]:
```python
# Check if the two outputs are equal
assert np.max(out_separable - out) < 1e-8
```