Homework 1

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.

In [1]:
```
# 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
```

Part 1: Convolutions

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: $f * h = h * f$).

Answer: Let $m - i = x$ and $n - j = y$,

$$
\begin{aligned}
(f*h)[m,n] &= \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f[i,j]\cdot h[m-i,n-j] && (3)\\
&= \sum_{x=-\infty}^{\infty}\sum_{y=-\infty}^{\infty} f[m-x,n-y]\cdot h[x,y] && (4)\\
&= \sum_{x=-\infty}^{\infty}\sum_{y=-\infty}^{\infty} h[x,y]\cdot f[m-x,n-y] && (5)\\
&= \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} h[i,j]\cdot f[m-i,n-j] && (6)\\
&= (h*f)[m,n] && (7)
\end{aligned}
$$

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 $g' = (f'*h)$.

Answer:

$$
\begin{aligned}
(f'*h) &= f[m-m_0,n-n_0]*h && (8)\\
&= \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} f[i,j]\cdot h[m-m_0-i,n-n_0-j] && (9)\\
&= g[m-m_0,n-n_0] && (10)\\
&= g'[m,n] && (11)
\end{aligned}
$$

Therefore, S defined by any kernel h is a shift invariant system.

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.

Answer:

$$
\begin{aligned}
S\{\alpha f_1[n,m] + \beta f_2[n,m]\} &= (\alpha f_1[n,m] + \beta f_2[n,m])*h && (12)\\
&= (\alpha f_1[n,m])*h + (\beta f_2[n,m])*h && (13)\\
&= \alpha(f_1[n,m]*h) + \beta(f_2[n,m]*h) && (14)\\
&= \alpha S\{f_1[n,m]\} + \beta S\{f_2[n,m]\} && (15)
\end{aligned}
$$

Therefore, S defined by any kernel h is linear.

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.

In [2]:
```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?

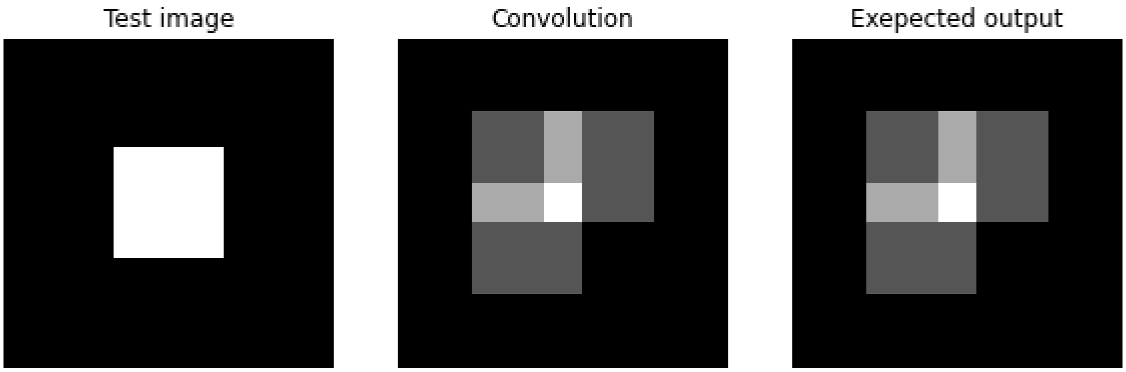
Now, implement the function **`conv_nested`** in **`filters.py`**. This is a naive implementation of convolution which uses 4 nested for-loops. It takes an image f and a kernel h as inputs and outputs the convolved image $(f * h)$ that has the same shape as the input image. This implementation should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

*- Hint: It may be easier to implement (hf)**

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

In [3]:

```
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."
```



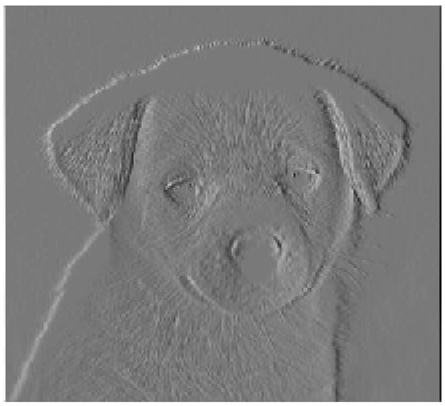
Now let's test your `conv_nested` function on a real image.

In [4]:

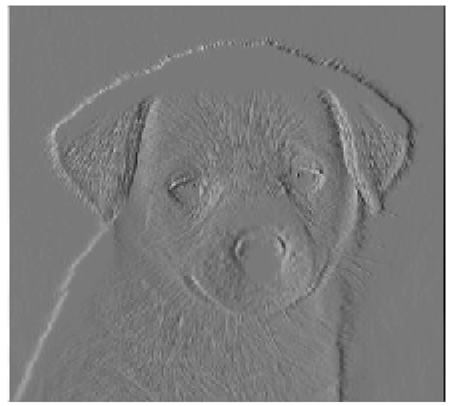
```
from filters import conv_nested

# 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

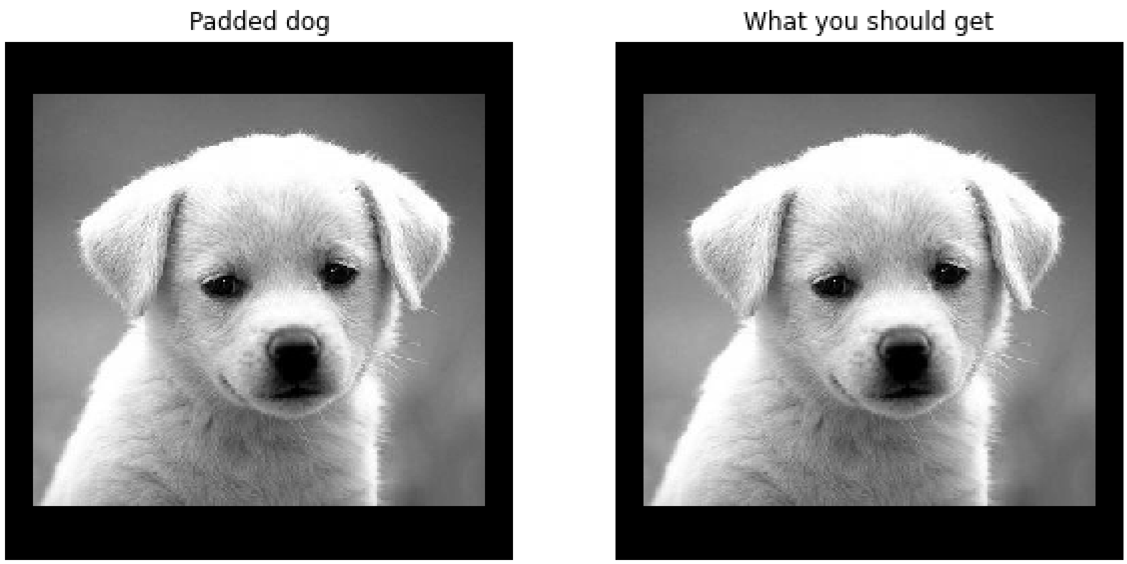


Let us implement a more efficient version of convolution using array operations in numpy. As shown in the lecture, a convolution can be considered as a sliding window that computes sum of the pixel values weighted by the flipped kernel. The faster version will i) zero-pad an image, ii) flip the kernel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

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

In [5]:

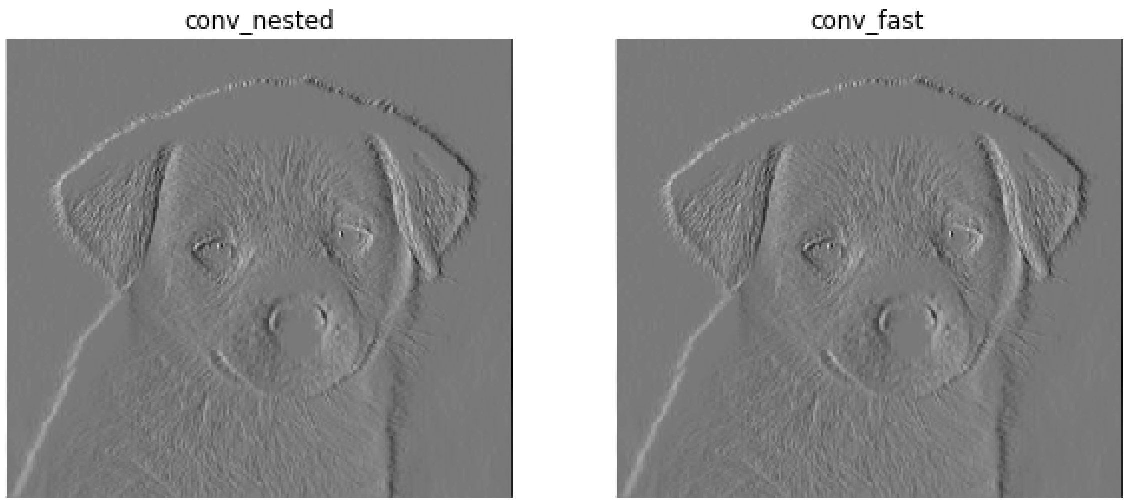
```
from filters import zero_pad

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` .

In [6]:
```
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))
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')

# 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 0.979618 seconds.
conv_fast: took 0.560964 seconds.
```



Part 2: Cross-correlation

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

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

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 CS131 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.

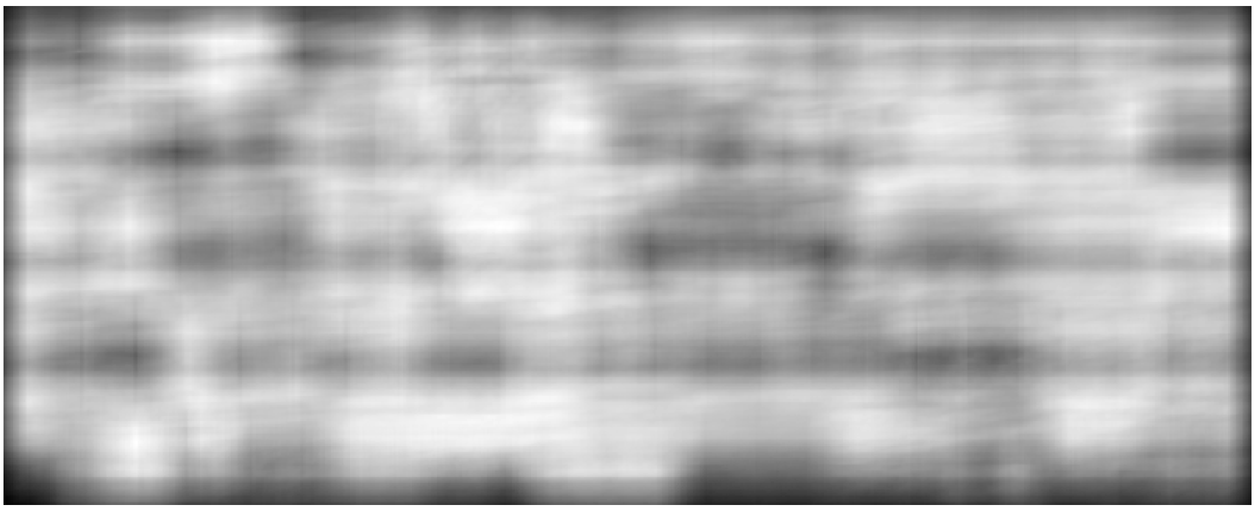
In [7]:

```
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)
```

```python
# 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)
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)

Interpretation

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

Answer: *The cross-correlation filter failed to detect the product correctly. Using the raw template, the resulting correlation value might be too high to be distinguishable because the maximum value for each pixel is 255.*

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.

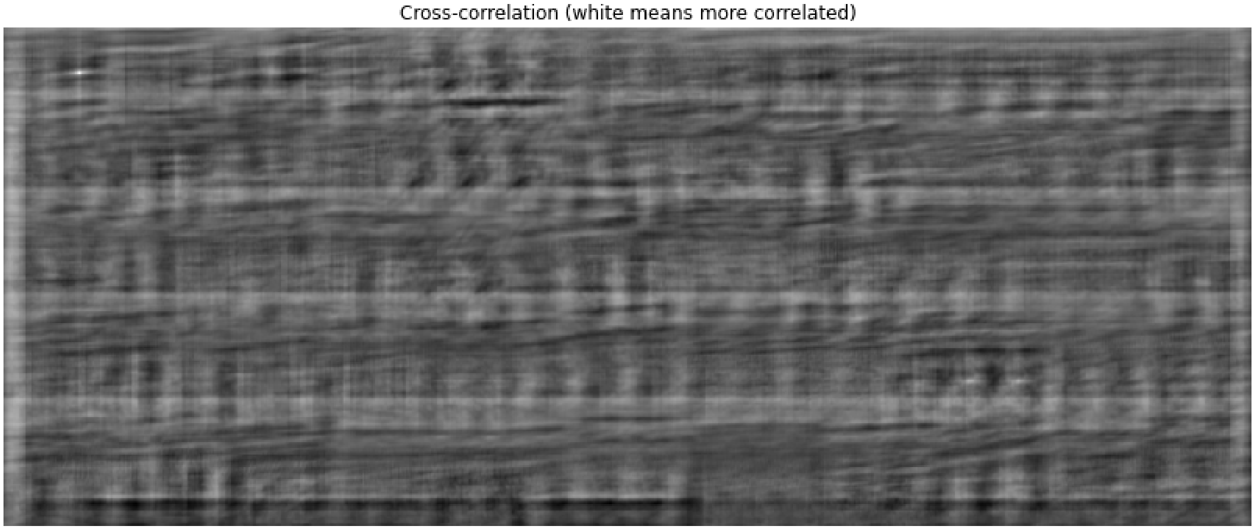
If your implementation is correct, you should see the blue cross centered over the correct cereal box.

In [8]:

```
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.

In [9]:
```
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 r
    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
```

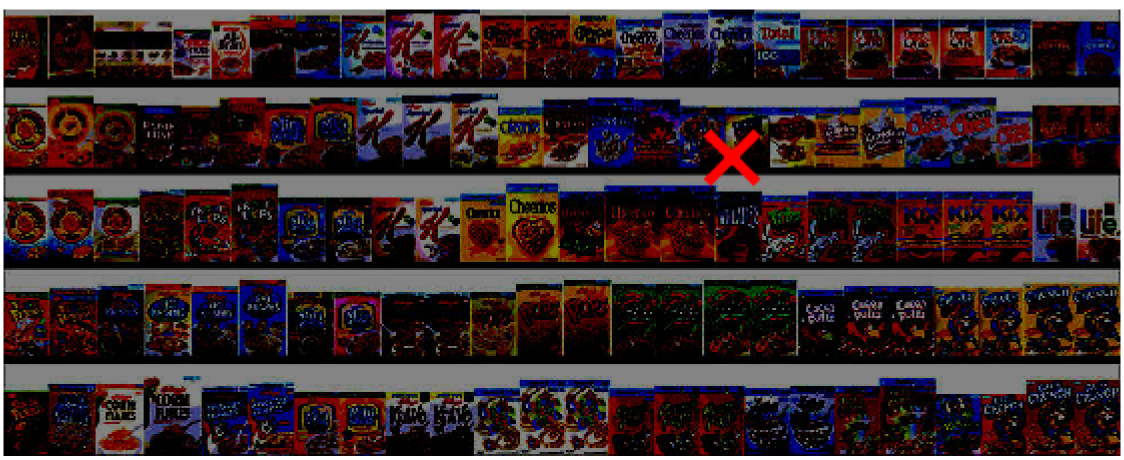
```
The product is not on the shelf
```

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.

In [10]:
```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 detcted location
plt.plot(x, y, 'rx', ms=25, mew=5)
plt.show()
```

Result (red marker on the detected location)



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:

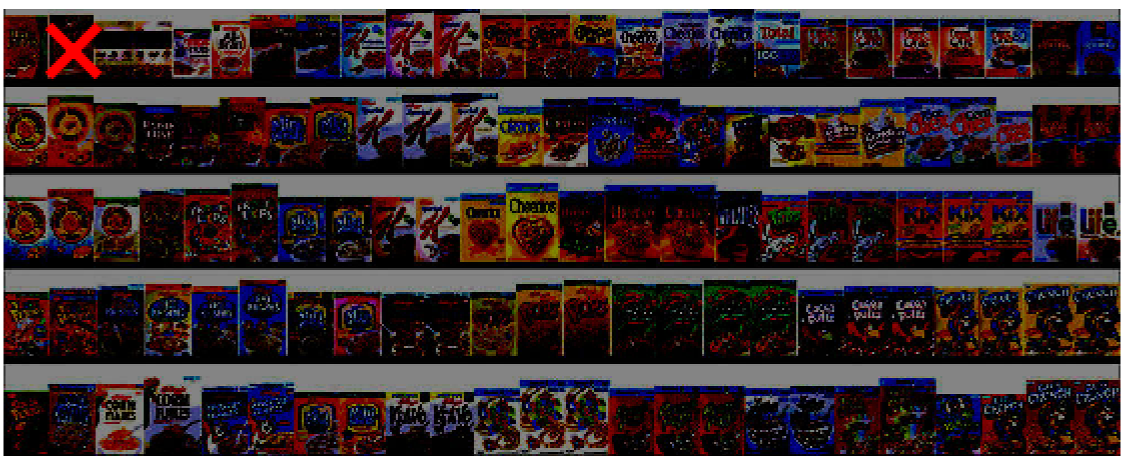
- $f_{m,n}$ is the patch image at position (m,n)
- $\overline{f_{m,n}}$ is the mean of the patch image $f_{m,n}$

- $\sigma_{f_{m,n}}$ is the standard deviation of the patch image $f_{m,n}$
- $\bar{g}$ 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.

In [11]:
```
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)



Part 3: Separable Filters

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 $F = F_1F_2$,

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

where $*$ is the convolution operation.

Answer:

$$\begin{aligned}
(I * F)[m, n] &= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I[i, j] \cdot F[m - i, n - j] && (16) \\
&= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I[i, j] \cdot F_1[m - i] \cdot F_2[n - j] && (17) \\
&= \sum_{j=-\infty}^{\infty} F_2[n - j] \sum_{i=-\infty}^{\infty} I[i, j] \cdot F_1[m - i] && (18) \\
&= \sum_{j=-\infty}^{\infty} F_2[n - j] \cdot (I * F_1) && (19) \\
&= (I * F_1) * F_2 && (20)
\end{aligned}$$

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. $F = F_1F_2$).

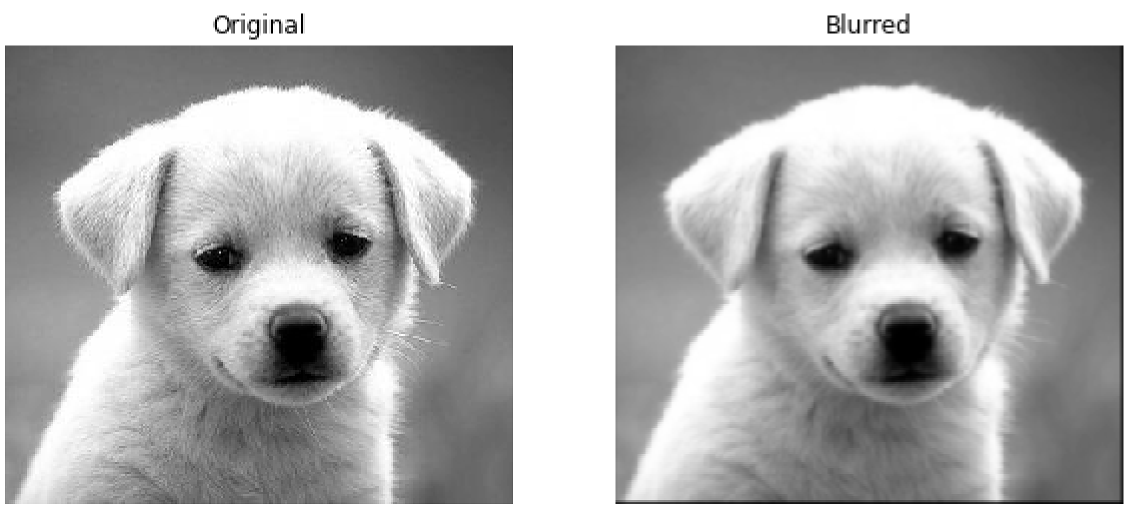
(i) How many multiplication operations do you need to do a direct 2D convolution (i.e. $I * F$)?
(ii) How many multiplication operations do you need to do 1D convolutions on rows and columns (i.e. $(I * F_1) * F_2$)?
(iii) Use Big-O notation, written with respet to the dimensions M_1, N_1, M_2, and N_2,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

Answer:
(i). $M_1 \times N_1 \times M_2 \times N_2$
(ii). $M_1 \times N_1 \times M_2 + M_1 \times N_1 \times N_2 = (M_2 + N_2) \times M_1 \times N_1$
(iii). The Big-O notation for 2D convolution can be written as $O(M_1N_1M_2N_2)$, while the Big-O notation for 1D convolution can be written as $O(M_1N_1)$. It's obvious that the two successive 1D convolution is more efficient than the direct 2D convolution.

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.

In [12]:
```
# 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.

In [13]:

```
# 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
k1 = np.array([[1],[4],[6],[4],[1]])
k2 = np.array([[1,4,6,4,1]])
### 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)"
```

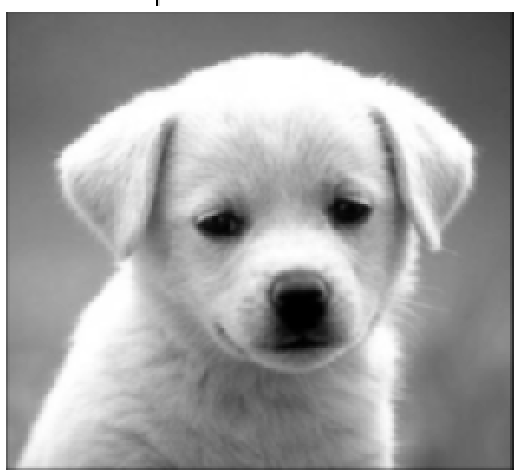
We now apply the two versions of convolution to the same image, and compare their running time. Note that the outputs of the two convolutions must be the same.

In [14]:

```
# 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

Separable convolution



```
Normal convolution: took 2.776247 seconds.
Separable convolution: took 1.218188 seconds.
```

In [15]:

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

In [ ]: