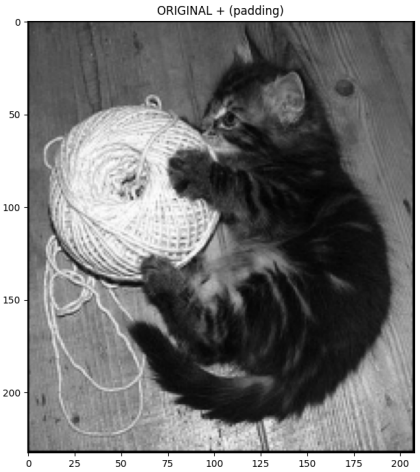
COMP37212 - Coursework 1: Convolution & Kernels

Image reading and padding:

First I have written a function to read the kitty.bmp file directly from the bytestrings, in python. I also made this function so that you can add a custom padding directly when the image is read, without needing to reiterate through it for padding, making it faster (For part 2 with the weighted mean average I have made a padding function for the additional convolution). It works with any nxn convolution kernel and any padding.

Convolution function for a nxn structural element:

Then I have written a convolution function from scratch, between an image and a size nxn structural element (kernel). It loops over all the image's pixels and applies the element (kernel) of



selected size to each one.

To make sure this is accurate when dealing with pixels on the edges and corners, I have the padding (border) for the image, pixels with value 0. I have also considered instead of giving values of 0, to give the values on the edges to the pad (border replication), as this could help in the convolution, but I have not implemented it.

I then also created a padding function to be able to apply both the weighted mean average and convolution functions.

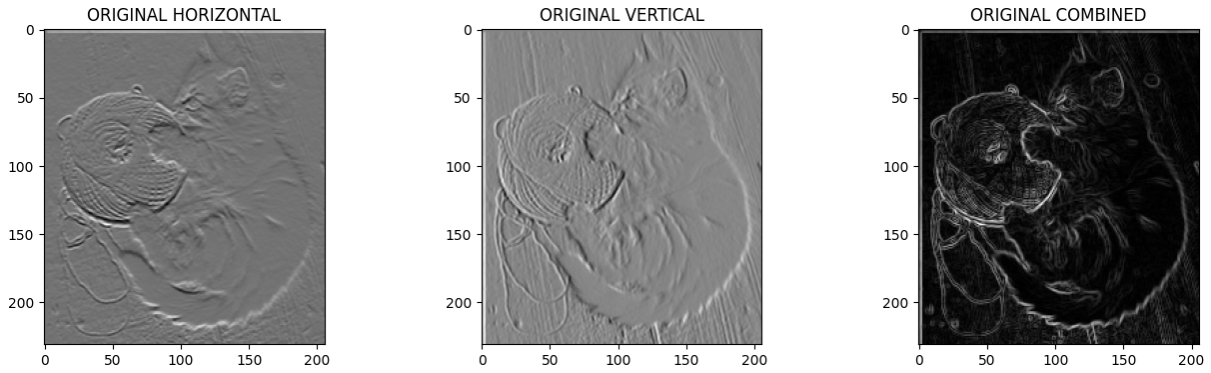
Horizontal and Vertical Gradient Images + combination of the 2:

For the Horizontal and Vertical gradient images I have tested 2 different elements setup, Sobel and Scharr.

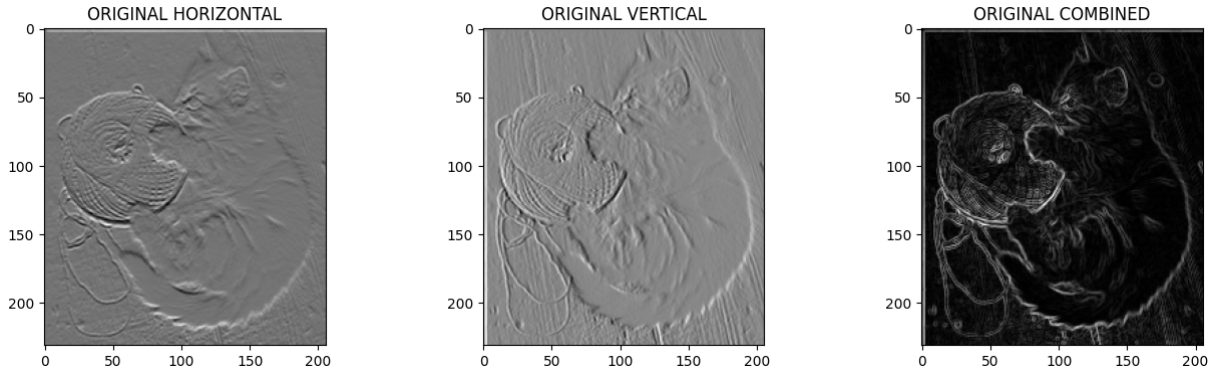
```
sobel kernel
horizontal
[[-1 -2 -1]
 [ 0  0  0]
 [ 1  2  1]]
vertical
[[-1  0  1]
 [-2  0  2]
 [-1  0  1]]
```

```
scharr kernel
horizontal
[[ -3 -10  -3]
 [  0   0   0]
 [  3  10   3]]
vertical
[[ -3   0   3]
 [-10   0  10]
 [ -3   0   3]]
```

Sobel



Scharr

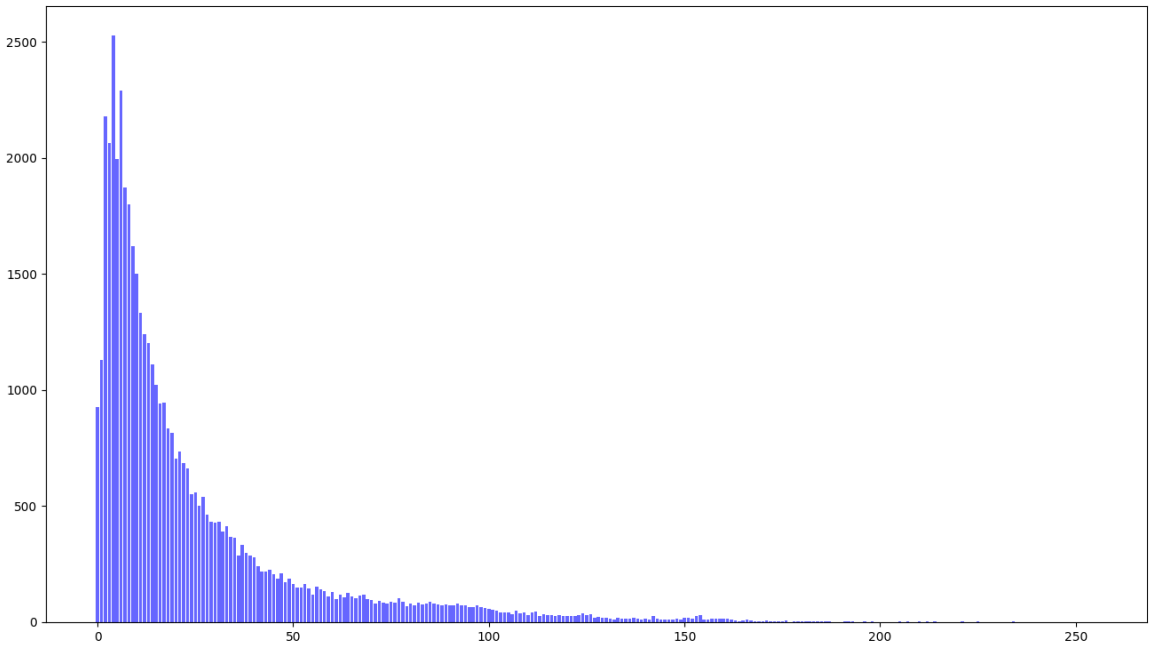


Through testing I have noticed that both the kernels provide good results and there is not a major difference noticed in the combined gradient images and the edges detected. For the purpose of this exercise I decided on using Sobel.

Then I implemented a thresholding function from scratch to detect the edges from the combined image of the horizontal and vertical gradients.

Histogram on the edge strength image:

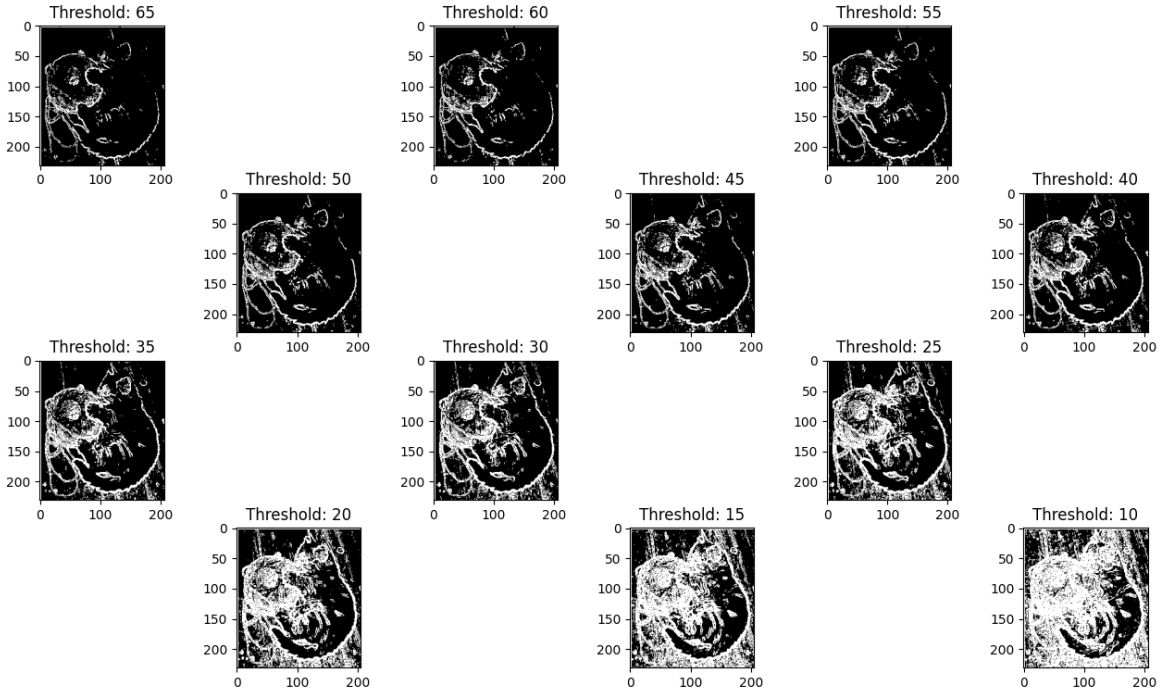
To find a good threshold I made use of the histogram from the edge strength image, and plotted it, to notice where the pixel value changed drastically.



Edge detection by thresholding image:

A good threshold would skip the peak and be where the “wave” starts to go down, for example between 20 and 50.

I have tested different thresholds to see the output for each:



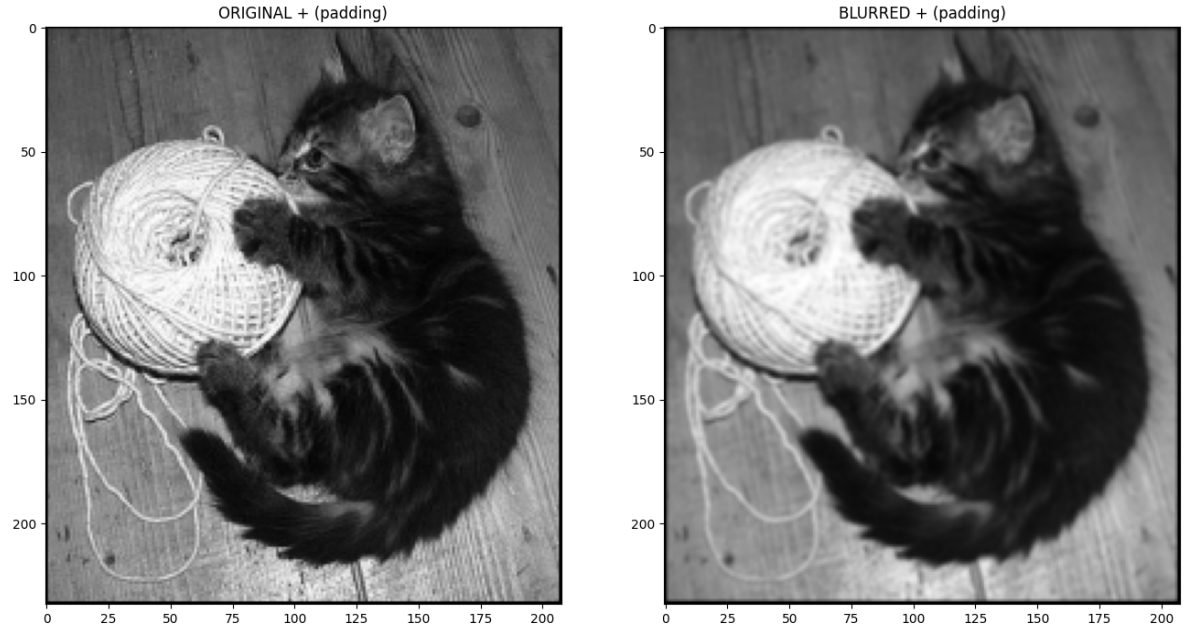
While results may seem good, at around 45-35, there is still a lot of noise and, for example, the fur of the cat and the wood still produce some edges that we would not want to take into consideration. If we get the threshold too high then the cat edges (“good edges”) start to disappear.

Repeated steps using weighted average kernel:

Next, I repeated those exact steps but additionally convoluted the original image first with the weighted mean average.

```
[[0.5, 1, 0.5],
 [ 1 , 2,  1 ],
 [0.5, 1, 0.5]]
```

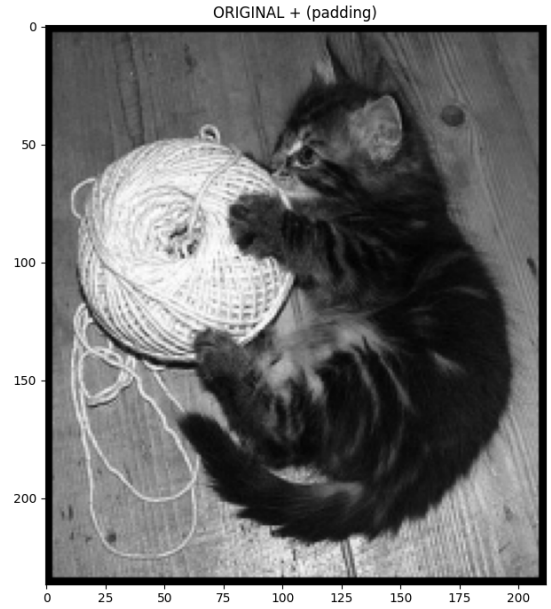
This resulted in a blurred version of the original image, which would theoretically work better for edge detection.

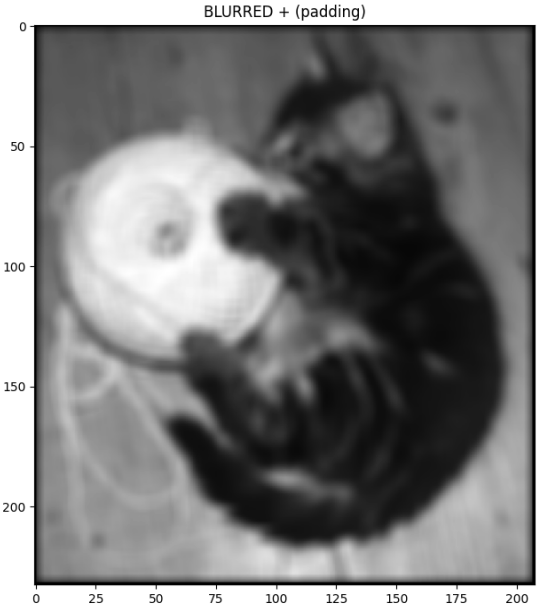


I have noticed improved results but then I decided I could use Gaussian smoothing. I made a kernel based on the Gaussian functions and then applied the respective kernel, with the help of the previously made convolution function, to the image, to receive a significantly improved blurred output.

This is the formula for the respective Gaussian kernel:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

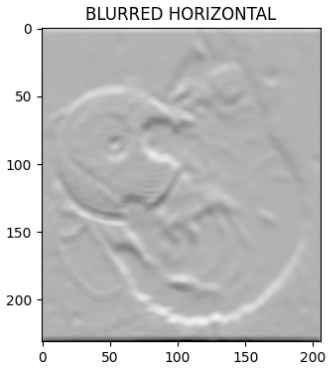


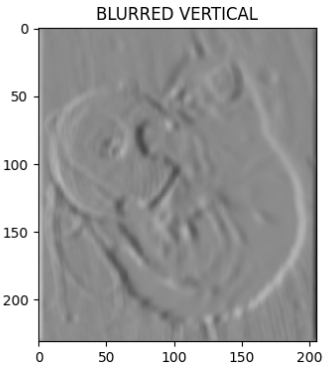


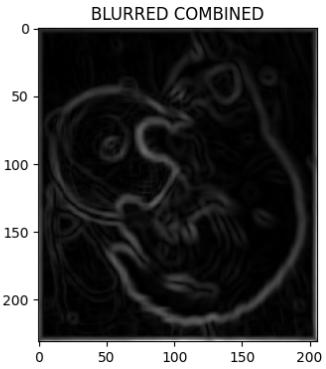
I have made a Gaussian Kernel of size 7x7 with a standard deviation of 3. Small changes on these hyperparameters do not make a distinguishable difference but could prove useful depending on what we want to do with the resulting image.

Conclusion:

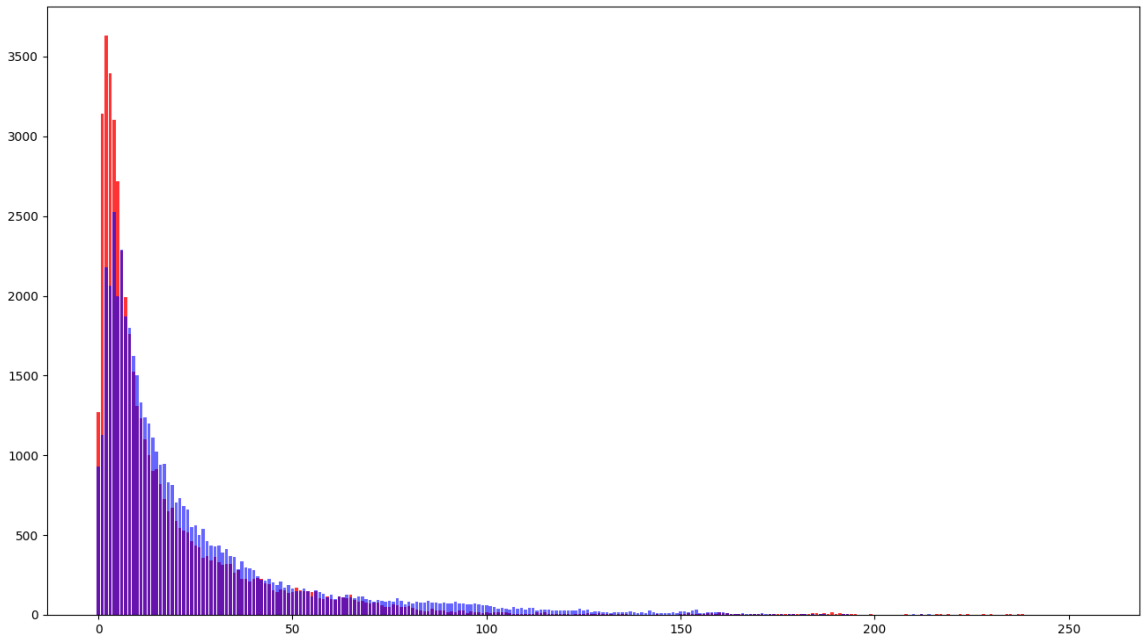
Already on the horizontal and vertical gradient images, plus the combined one, we can notice a big improvement that the weighted mean average (blur) has achieved.



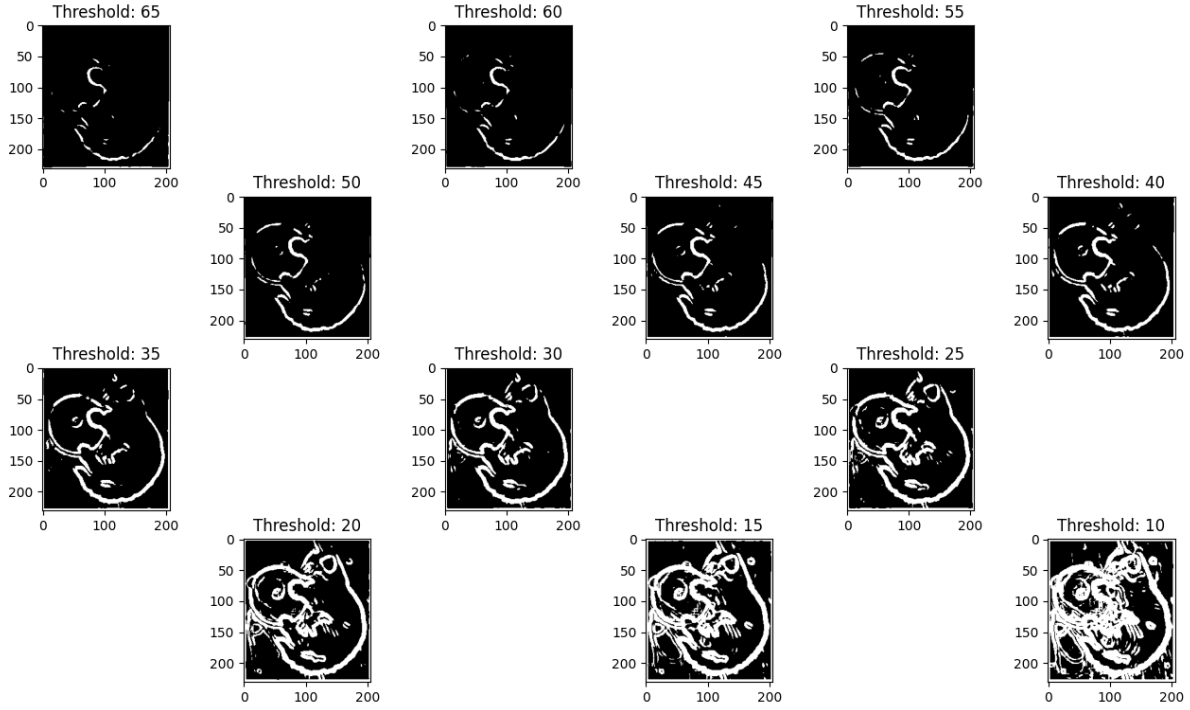




Also, I have gotten a different histogram, with a little more abrupt curve and a bigger difference between the 0 and 255 pixels, compared to the original one, which makes it much better for the thresholding (BLUE is for the original image and RED is for the blurred image):



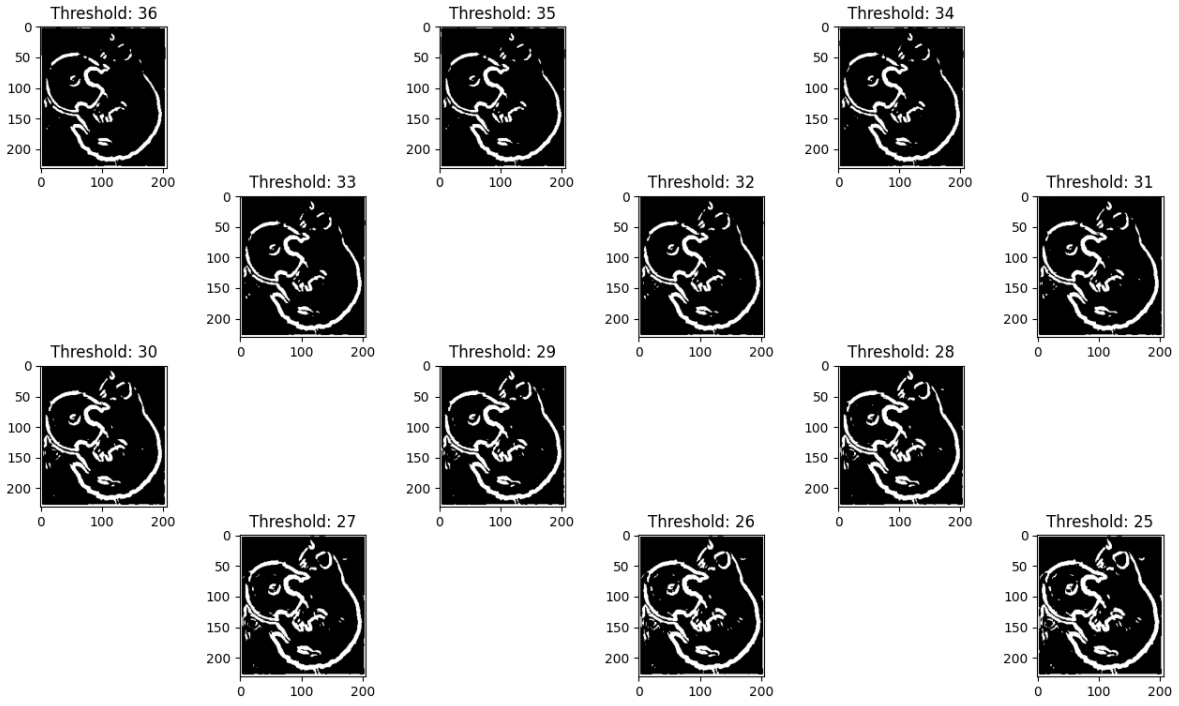
Testing the threshold edge detection on the edge strength image resulted, I have gotten significantly better results with various thresholds:



From the histogram and experiment images, we can deduce that a threshold lower than 20 would show too many edges and a lot of noise in the wood floor and cat's fur. If we go above 40 we notice that significant edges start to disappear, as in the cat's head and top part.

So, personally I would go with a threshold between 35 and 25. If we go lower we will need to introduce additional techniques to remove noise, for example, removal of sole or alone edges and points that most probably do not resemble an edge. If we go higher we will need to connect detected edges to compensate for the lost ones.

I also displayed images with the thresholds that I have stated above, from which we can pick the best one to use and decide on possible additional processing of the edges to get a better result. The best in my opinion would be somewhere between 29 and 32



Comparing the two approaches, it is clear that the results from the second experiment are much better than from the first (without blur). The gaussian kernel removes a significant amount of

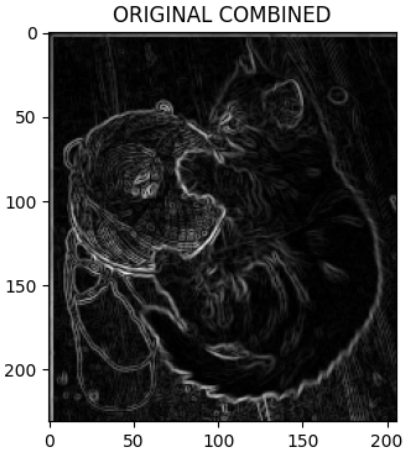
noise and gets rid of insignificant edges that we would not want to detect. The blurred image has more defined edges and reduced noise caused from background elements, as we can see the bottom right corner where edges formed on the wood grain, or in the twine ball where a lot of edges formed inside it, or small spots in the fur of the cat.

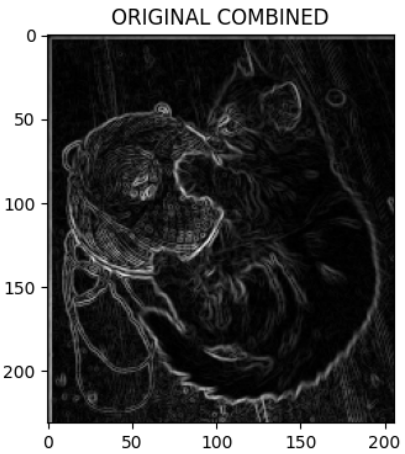
Thus, without doubt, the second approach, introducing a weighted average kernel (gaussian blur), gives a better result.

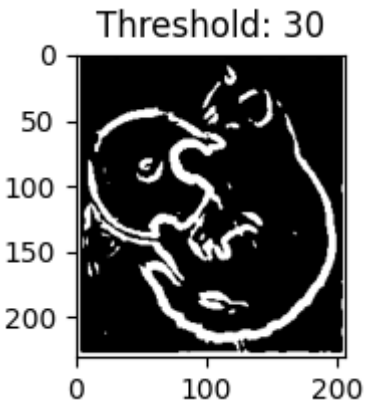
Extra:

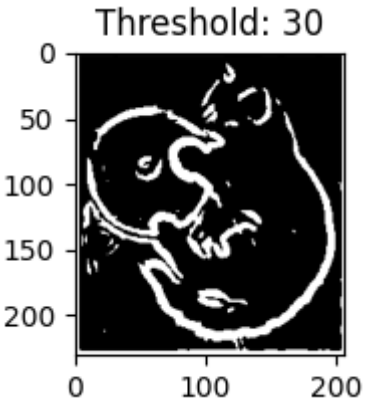
I managed to read the kitty.bmp file directly, through its bytestrings, and with this I was able to add padding as the image was being read, saving a call to the padding function. This also helped save memory as instead of reading the images 3 colours channels per pixel, I read it directly as grayscale, only reading 1 value from that bytestring.

I tried using two different kernels for the gradient magnitude image, Sobel and Scharr but the difference in the end result was not distinguishable. (1. Sobel - 2. Scharr)

1. 

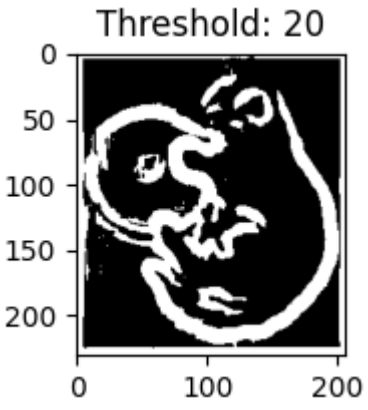
2. 

1. 

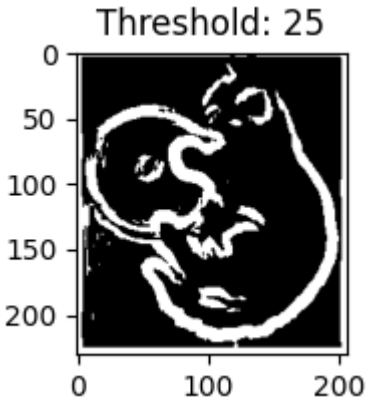
2. 

While the Gaussian kernel performed much better than a normal weighted mean average, I was still not satisfied with the detected edges so I decided to experiment with the kernel size and standard deviation of the Gaussian function.

Increasing the kernel size of the Gaussian filter leads to a better result, with less noise, but at the cost of making the edges much bigger (example with kernel size 11x11 and standard deviation 7:

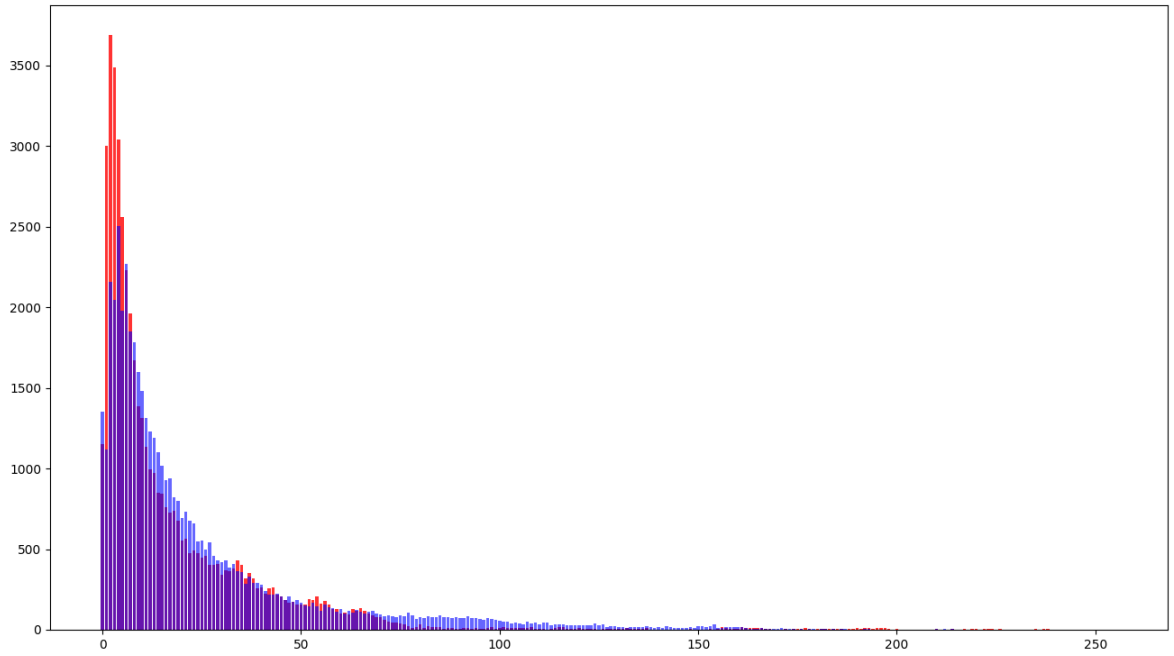


Using a smaller kernel of 9x9 and a bigger standard deviation of 10 gives us a nicer edge detection, less noise but defined edges:



While this might not be perfect or might need more processing, it is one of the best results I've gotten.

This can also be deducted from the histogram in which we notice the difference in the wave of the pixels 0-255 and how much more initial pixels there are from 0 to 15:



In conclusion, we clearly need to add blurring of the image to get a consistent edge detector, the two main parameters here are the threshold for edges and the blur kernel (Gaussian kernel) size and standard deviation. I got a satisfactory result with a kernel size of 9x9, albeit 7x7 and 11x11 were not bad either but this one performed a little bit better, and the standard deviation helped a lot in removing noise and making the edge detection smoother, 1-3-5 were to small but over 20 edges started to get too big, thick and bulky.

I have used matplotlib for plotting the histogram and because OpenCV needed more matrix manipulation, I decided on using matplotlib also for displaying the resulting images. I also used python math to implement the Gaussian Kernel.

CODE:

```
from matplotlib import pyplot as plt
import math

def read_image_and_pad(path, width, height, padding):
    """ Read .bmp file and pad directly """
    # Calculate starting bit (multiple of 4)
    if width%4 == 0:
        width_offset = width
```

```
else:
    width_offset = width+(4-width%4)
# Skip offset of the .bmp file input bytestring
file = open(path, "rb")
file.seek(10)
start = int.from_bytes(file.read(4), "little")
file.seek(18)
file.read(8)
file.seek(start-1)

img = []
line = []
index = 0
for _ in range(padding):
    line = []
    for _ in range(width + 2*padding):
        line.append(0)
    img.insert(0, line)
line = []
for _ in range(padding):
    line.append(0)
while True:
    if index == width_offset:
        index = 0
        for _ in range(padding):
            line.append(0)
        img.insert(0, line)
        line = []
        for _ in range(padding):
            line.append(0)
    index += 1
    bit_string = file.read(1)

    if len(bit_string) == 0:
        for _ in range(padding):
            line = []
            for _ in range(width + 2*padding):
                line.append(0)
            img.insert(0, line)
        break
    if index < 4-width%4 or index >= width_offset:
        continue

    pixel = ord(bit_string)
    line.append(pixel)

file.close()
```

```
    return img

def padd_img(img, width, height, padding):
    """ Return input image padded"""
    padded_img = []
    for _ in range(padding):
        padded_img.append([0 for _ in range(width)])

    padded_img.extend(img.copy())
    padded_img = [x+ [0]*padding for x in padded_img]
    padded_img = [[0]*padding +x for x in padded_img]
    for _ in range(padding):
        padded_img.append([0 for _ in range(width+2*padding)])

    return padded_img

def convolution(img, width, height, padding, kernel):
    """ Convolution function on already padded image and input kernel
    Need to enter pad number and kernel"""
    kernel_size = len(kernel)
    kh = len(kernel)
    kw = len(kernel[0])
    h = height - kh + 2 * padding + 1
    w = width - kw + 2 * padding + 1
    img_out = [
        [0 for _ in range(w)]
        for _ in range(h)
    ]

    # Find the highest value in the matrix for normalization
    max_value = 0
    for i in range(h):
        for j in range(w):
            for k in range(kernel_size):
                for l in range(kernel_size):
                    img_out[i][j] += (img[i + k][j + l] * kernel[k][l])
            max_value = max(max_value, img_out[i][j])

    return img_out, max_value

def gradient_compute(img, kernel_name, width, height, padding):
    """ Compute the horizontal and vertical gradient images
    and then do the gradient magnitude of them
    (combine - square root of sum of squares)
    Two kernels to choose from """
    if kernel_name == "sobel":
```

```
        # Sobel Kernels
        kernel_horizontal = list()
        kernel_horizontal.extend([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
        kernel_vertical = list()
        kernel_vertical.extend([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    elif kernel_name == "scharr":
        # Scharr Kernels
        kernel_horizontal = list()
        kernel_horizontal.extend([[-3, -10, -3], [0, 0, 0], [3, 10, 3]])
        kernel_vertical = list()
        kernel_vertical.extend([[-3, 0, 3], [-10, 0, 10], [-3, 0, 3]])
    else:
        exit()

    # Apply the convolution function to the image both horizontally and
vertically
    img_horizontal, max_1 = convolution(img, width, height, padding,
kernel_horizontal)
    img_vertical, max_2 = convolution(img, width, height, padding,
kernel_vertical)

    # Calculate the gradient magnitude between horizontal and vertical
gradient images
    gradient_magnitude = magnitude_compute(img_horizontal, img_vertical)

    for i in range(len(img_horizontal)):
        for j in range(len(img_horizontal[0])):
            img_horizontal[i][j] /= max_1

    for i in range(len(img_vertical)):
        for j in range(len(img_vertical[0])):
            img_vertical[i][j] /= max_2

    max_val = 0
    for i in range(len(gradient_magnitude)):
        for j in range(len(gradient_magnitude[0])):
            max_val = max(max_val, gradient_magnitude[i][j])

    # Normalize the image to from 0-1 to 0-255
    for i in range(len(gradient_magnitude)):
        for j in range(len(gradient_magnitude[0])):
            gradient_magnitude[i][j] = int(gradient_magnitude[i][j] /
(max_val * 1/255))

    return gradient_magnitude, img_horizontal, img_vertical

def magnitude_compute(img1, img2):
```

```
    """ Combine two images """
    img_mag = [
        [0 for _ in range(len(img1[0]))]
        for _ in range(len(img1))
    ]

    for i in range(len(img1)):
        for j in range(len(img1[0])):
            img_mag[i][j] = math.sqrt(img1[i][j] ** 2 + img2[i][j] ** 2)

    return img_mag

def histogram(img, width, height):
    """ Return a histogram of a grayscale image (0-255) """
    histogram = [0 for _ in range(256)]
    for i in range(height):
        for j in range(width):
            histogram[img[i][j]]+=1
    return histogram

def threshold(img, width, height, threshold):
    """ Edge detection thresholding function """
    h = height
    w = width
    img_out = [
        [0 for j in range(w)]
        for i in range(h)
    ]

    for i in range(h):
        for j in range(w):
            if (img[i][j] >= threshold):
                img_out[i][j] = 255
            else:
                img_out[i][j] = 0

    return img_out

def gaussian_blur(kernel_size, std_dev):
    """ Return a gaussian blur kernel based on the product of two
Gaussian functions """
    kernel = [
        [0 for _ in range(kernel_size)]
        for _ in range(kernel_size)
    ]
    for i in range(kernel_size):
        for j in range(kernel_size):
```

```
            kernel[i][j] = (1/(2 * math.pi * std_dev**2)) *
math.e**(-(((j - kernel_size//2)**2 + (i - kernel_size//2)**2) / (2 *
std_dev**2)))
    return kernel

def weightead_mean(img, width, height, padding, name, kernel_size,
std_dev):
    """ Apply a blur on the image (gaussian or other) """
    if (name=="gaussian"):
        kernel = gaussian_blur(kernel_size, std_dev)
    else:
        kernel = list()
        kernel.extend([[1, 2, 1], [2, 4, 2], [1, 2, 1]])

    img, max_val = convolution(img, width, height, padding, kernel)

    for i in range(len(img)):
        for j in range(len(img[0])):
            img[i][j] /= max_val

    return img

# -- PARAMETERS --
path = "kitty.bmp"
width = 206
height = 231

kernel_name = "sobel"
kernel_blur = "gaussian"

gaussian_kernel_size = 9
gaussian_std_dev = 10


padding_blur = gaussian_kernel_size // 2
padding_gradient = 1

thresholds_original = [10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65]
#thresholds =          [10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65]
thresholds =          [15, 18, 20, 21, 23, 24, 25, 27, 29, 30, 32, 34]

# -- READ IMAGE AND MAKE PADDING --
img_original = read_image_and_pad(path, width, height, padding_blur)

# -- WEIGHTED MEAN FIRST ON ORIGINAL IMAGE --
#img = padd_img(img, width, height, padding)
```

```
img_blur = weightead_mean(img_original, width, height, padding_blur,
kernel_blur, gaussian_kernel_size, gaussian_std_dev)
img_blur = padd_img(img_blur, width, height, padding_gradient)

# -- CALCULATE GRADIENT IMAGES --
gradient_magnitude, img_horizontal, img_vertical =
gradient_compute(img_blur, kernel_name, width, height,
padding_gradient)
gradient_magnitude_original, img_horizontal_original,
img_vertical_original = gradient_compute(img_original, kernel_name,
width, height, padding_gradient)
#gradient_magnitude = gradient_compute(img, "scharr", True)
#img_original = gradient_magnitude.copy()

# -- PERFORM EDGE DETECTION BASED ON SET THRESHOLD --
imgs_final_original = list()
for i in thresholds_original:
   imgs_final_original.append(threshold(gradient_magnitude_original,
width, height, threshold=i))
#thresholds_original.reverse()

imgs_final = list()
for i in thresholds:
   imgs_final.append(threshold(gradient_magnitude, width, height,
threshold=i))
#thresholds.reverse()

# -- PLOT ALL THE DATA --

fig, ax = plt.subplots(1, 2, figsize=(16, 9))
ax[0].imshow(img_original, cmap="gray")
ax[0].set_title("ORIGINAL + (padding)")
ax[1].imshow(img_blur, cmap="gray")
ax[1].set_title("BLURRED + (padding)")
plt.show()

fig, ax = plt.subplots(2, 3, figsize=(16, 9))
ax[0, 0].imshow(img_horizontal_original, cmap="gray")
ax[0, 1].imshow(img_vertical_original, cmap="gray")
ax[0, 2].imshow(gradient_magnitude_original, cmap="gray")
ax[0, 0].set_title("ORIGINAL HORIZONTAL")
ax[0, 1].set_title("ORIGINAL VERTICAL")
ax[0, 2].set_title("ORIGINAL COMBINED")
ax[1, 0].imshow(img_horizontal, cmap="gray")
ax[1, 1].imshow(img_vertical, cmap="gray")
ax[1, 2].imshow(gradient_magnitude, cmap="gray")
ax[1, 0].set_title("BLURRED HORIZONTAL")
```

```
ax[1, 1].set_title("BLURRED VERTICAL")
ax[1, 2].set_title("BLURRED COMBINED")
plt.show()

# -- PLOT THE HISTOGRAM --
bar_height = [x for x in range(256)]
histogram_original = histogram(gradient_magnitude_original, width,
height)
histogram = histogram(gradient_magnitude, width, height)
fig, ax = plt.subplots(1, 1, figsize=(16, 9))
ax.bar(bar_height, histogram, color='r', alpha=0.8)
ax.bar(bar_height, histogram_original, color='b', alpha=0.6)
plt.show()

x = 0
for edges in [imgs_final_original, imgs_final]:
   fig, ax = plt.subplots(4, 6, figsize=(16, 9))
   if x == 0:
       fig.suptitle("ORIGINAL")
   else:
       fig.suptitle("BLURRED")
   for i in range(4):
       for j in range(6):
           if(i+j)%2 == 1:
               ax[i, j].set_visible(False)
               continue
           if x == 0:
               ax[i, j].set_title("Threshold: " +
str(thresholds_original.pop()))
           else:
               ax[i, j].set_title("Threshold: " +
str(thresholds.pop()))
           ax[i, j].imshow(edges.pop(), cmap="gray")
   plt.show()
   x+=1
```