COMP37212 Coursework 1 - Convolution and Kernels

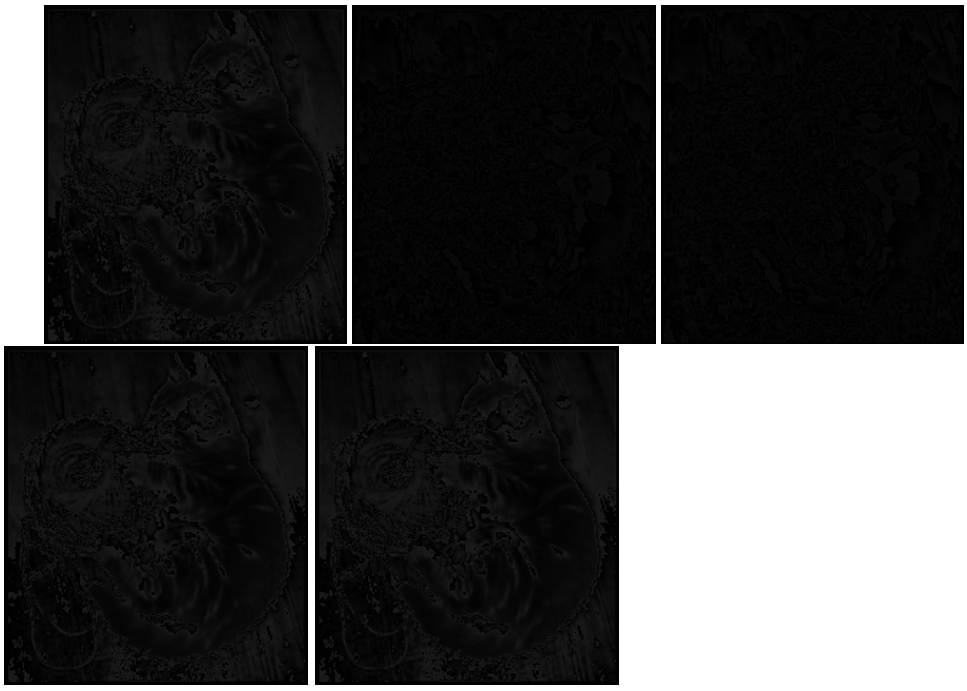
Cameron Groves

March 2021

Task 1: I initially wanted to compare the difference in different sized structuring elements. Mainly 3x3 or 5x5 and a test of 7x7 which did seem to provide the best image. So overall the effects of the 7x7 element were much better compared to the 5x5 and 3x3s however there was a visible change in time to process the image making it far less practical in a real world scenario. The images were very similar to look at so to truly compare them it was better to use difference images to see clear changes in the images. Overall the best effects came from a 5x5 element as it best balanced speeds and had an improved final image.

I also experimented with variations of weightings some kernels have, like the standard [1,1,1] compared with weighted [1,2,1] and some other alternatives. Once again to effectively compare the methods we needed difference images. Overall when only comparing weights the weighted or gaussian method definitely provided an advantage over averaging but for the weighted methods both seem to have their advantages depending on the situation. In this context with a kitten with an excess of fur the image requires either reducing or smoothing alongside, like gaussian.

Considering the effects of applying convolution filters, the same or different, more than once and the combination the achieve. After a single step of processing i found the image was still not achieving a good image, investigating multiple kernels. Overall the better methods were a non smoothing kernel and either a reduction of the image or apply a smoothing filter, both of which give good results and to apply them both creates a good image for edge detection.



```
def check_image(name):
    img = cv2.imread(name, 0) # read and check image
    if img is None:
        print("error")
return img

def pad_image(img, struc_size):
# pad image with (struc size / 2) zeros
    pad_size = struc_size // 2
    height, width = img.shape[:2]
    padded_img = np.zeros(shape=(height+(2*pad_size), width+(2*pad_size))
    , dtype=np.uint8)
    for i in range(0, height):
        for j in range(0, width):
            padded_img[i+pad_size, j+pad_size] = img[i,j]
    return padded_img

def reduce_image(self, img):
    height, width = img.shape[:2]
    reduced_image = np.zeros(shape=(height/2,width/2), dtype=np.uint8)
    for i in range(0, height/2, 2):
        for j in range(0, width/2, 2):
            reduced_image[i/2,j/2] = img[i,j]
    return reduced_image
```

Task 2: Next was detecting Vertical and Horizontal edges and a brief assessment of the prewitt vs sobel. Overall the differences were minimal but prewitt provides a slight advantage however the difference images don't visibly have much change overall for a quick process of the image. The prewitt kernel is however the better one, for almost all other tests prewitt will be used.

The effects of reducing an images size and smoothing before edge detection and whether is provides a smoother edge. Smoothing or Blurring applied to the image results in a larger edges being less blurred than the smaller ones, resulting in better edges. Reducing the image however makes the edges smaller so more limited edges can be passed over by the kernel.

```
def convolve_image(img, kernel, struc_size):
    height, width = img.shape[:2]
    convoluted_image = np.zeros(shape=(height, width), dtype=np.uint8)
    pad_size = struc_size // 2
    for i in range(1, height-1):
        for j in range(1, width-1):
            for x in range(-pad_size, pad_size):
                for y in range(-pad_size, pad_size):
                    convoluted_image[i,j] += (img[i+x,j+y]*kernel[x,y])
            convoluted_image[i,j] = convoluted_image[i,j] / kernel.size
    return convoluted_image

def straight_edges(img, detector):
    height, width = img.shape[:2]
    straight_image = np.zeros(shape=(height, width), dtype=np.uint8)
    detector_size = int(math.sqrt(detector.size) / 2)
    for i in range(1, height-1):
        for j in range(1, width-1):
            for x in range(-detector_size, detector_size):
                for y in range(-detector_size, detector_size):
                    straight_image[i,j] += (img[i+x,j+y]*detector[x,y])
    return straight_image

def combine_edge_strength(horizontal_image, vertical_image):
    height, width = horizontal_image.shape[:2]
    combined_image = np.zeros(shape=(height, width), dtype=np.uint8)
    for i in range(1, height-1):
        for j in range(1, width-1):
            new_value = math.sqrt(horizontal_image[i,j]**2 +
            vertical_image[i,j]**2)
            combined_image[i,j] = new_value
    return combined_image
```

Task 3: Initially i started with evaluating the histograms of the images before i began thresholding to see if there was a disernable pattern. Given the colours

of the image and overall consistent darkness of the image makes almost all the histograms quite useless.

So for the thresholding method, i used OTSU and binary or a combination to experiment with. Effective values for thresholding a standard processed image seemed to be best thresholded between 50-70 these were found through trackbars and experimentation and OTSU auto thresholding led to a similar result. The style and quality of thresholding slightly depended on the steps beforehand.

Lastly thresholding typically is better after processing however given the kitten in initial image is already so dark compared to most of the image that we can threshold straight away and receive a decently thresholded image. That image can be further processed with the initial methods to produce one of the better final images.

```
def on_change_thresh(value):
    #T, img_threshold = cv2.threshold(img_combine, value, 255, cv2.THRESH_BINARY
    + cv2.THRESH_OTSU)
    ret, img_threshold = cv2.threshold(img_threshold, value, 255, cv2.THRESH_BINARY)
    #cv2.namedWindow('Thresholded')
    cv2.imshow('Thresholded', img_threshold)

#T, img_threshold = cv2.threshold(img_combine, 125, 255, cv2.THRESH_BINARY
+ cv2.THRESH_OTSU)
ret, img_threshold = cv2.threshold(img_combine, 125, 255, cv2.THRESH_BINARY)
cv2.imshow('Thresholded', img_threshold)
cv2.createTrackbar('slider', 'Thresholded', 0, 255, on_change_thresh)

def histogram(img, name):
# Calculate the histogram
    hist = cv2.calcHist([img], [0], None, [256], [0, 256])
    hist = hist.reshape(256)

    # Plot histogram
    plt.bar(np.linspace(0,255,256), hist)
    plt.title(name)
    plt.ylabel('Frequency')
    plt.xlabel('Grey Level')
    plt.show()
```