Computational Neuroscience

EEE 482/582

Can Kocagil

21602218

Homework-2

Department of Electric & Electronics Engineering

Bilkent University

Ankara, Turkey

1.03.2021

Table of Contents

List of Figures

1. QUESTION 1

In this question, the responses of cat LGN cell to two-dimensional visual images are given, data are described in Kara et al., Neuron 30:803-817 (2000). In the data, counts is a vector containing the number of spikes in each 15.6 ms bin, and stim contains the 32767, 16x16 images that were presented at the corresponding times.

1.1. Part A.

In part a, we are asked to calculate the STA images for each of the 10 time steps before each spike and show them all with a gray-scale colormap and identical display windowing. Then, we need to discuss STA derived filter and its spatio-temporal stimulus this LGN cell is selective for.

The spike-triggered average (STA) is a measure to relate a continuous signal and a simultaneously recorded spike train [1]. It represents the average signal taken at the times of spike occurrences and with proper normalization is equivalent to the cross-correlation between the continuous signal and the spike train [1]. The STA provides an estimate of a neuron's linear receptive field [4]. From the mathematical perspective, STA is the averaging operation on stimulus preceding a spike and it starts by the computing time window preceding each spike in spike train, then spike-triggered stimuli is averaged. Statistically, STA is unbiased estimator of neuron's receptive field if the stimulus distribution is spherically symmetric (e.g., Gaussian white noise) [4]. Let's proceed by constructing STA in algorithmic way.

Let $\mathbf{x_i}$ denote the spatio-temporal stimulus vector, with i^{th} time bin and let $\mathbf{y_i}$ is the spike count corresponds to that particular bin. Hence, STA can be calculated as follows

$$STA = \frac{1}{n_{sp}} \sum_{i=1}^{T} y_i x_i \ \ where \ n_{sp} = \sum_{i=1}^{T} y_i \tag{1}$$

Hence, in matrix formulation, let $\mathbf{X}$ denote the matrix whose rows are x_i^T then let y is the corresponding spike count whose i^{th} element is y_i so that

$$X = \begin{pmatrix} x_1^T \\ \vdots \\ x_n^T \end{pmatrix} \ \ and \ y = \begin{pmatrix} y_i \\ \vdots \\ y_n \end{pmatrix} \implies STA = \frac{1}{n_{sp}} X^T y \tag{2}$$

Then, let's apply the STA algorithm in Python using the **NumPy**. Here **SciPy** is imported for loading **.mat** file into Python environment and **Matplotlib** for data visualization.

```
import numpy as np
import scipy.io as io
import matplotlib.pyplot as plt
```

Then, the data **c2p3.mat** is parsed to extract spike-train (count) and stimulus data.

```
data = io.loadmat('c2p3.mat')
counts = data['counts']
stim = data['stim']

print(f"The shape of the counts data {counts.shape}")
print(f"The shape of the Stimilus data {stim.shape}")
```

The shape of the counts data (32767, 1)
The shape of the Stimilus data (16, 16, 32767)

Hence, as provided, we have a dataset of 32767 16x16 gray scale stimilus data with corresponding spike-trains. Here, I calculated the STA in Python.

```python
def STA(stim:np.ndarray,rho:np.ndarray,num_timesteps:int) -> np.ndarray:
    """
        Given the stimilus, rho (spike train) and number of time steps, computes
        Spike-Triggered-Average (STA). The spike-triggered average (STA) is a
↪   measure to relate a continuous signal and a simultaneously recorded spike
↪   train. It represents the average  signal  taken at the times of spike
↪   occurrences and with proper normalization is equivalent to the
↪   cross-correlation between the continuous signal and the spike train. The STA
↪   provides an estimate of a neuron's linear receptive field.

            Arguments:
                - stim (np.ndarray)   : Stimulus data to be subjected
                - rho  (np.ndarray)   : Time-series Spike-train data
                - num_timesteps (int) : Number of timesteps before spike

            Returns:
                - _STA (np.ndarray)    : Averaged Stimulis data taken at spike
↪   occurrences

    """

    # Creating zero matrix for STA:
    stim_h,stim_w = stim.shape[:2]
    _STA = np.zeros((stim_h,stim_w,num_timesteps))

    # Finding spike times + num_timesteps
    spike_times = rho[num_timesteps:].nonzero()[0] + num_timesteps

    print(f'There are {len(spike_times)} of spikes.')

    for idx_spike in spike_times:
        _STA += stim[:,:,idx_spike - num_timesteps:idx_spike]
    _STA = _STA.astype(np.float)
    _STA /= len(spike_times)

    return _STA
```

Then, number of time steps before each spike is provided as 10. So, the next step is plot the distribution of STA for last 10 steps before spike. Here is the Python code for plotting STA images for each of the 10 time steps before each spike in gray scale color-map and identical display.

```python
num_timesteps = 10
sta = STA(stim,counts,num_timesteps)

kwargs = dict(
    cmap = 'gray',
    vmin = sta.min(),
    vmax = sta.max()
)

```

```
10
11 for i in np.arange(num_timesteps):
12     plt.figure()
13     plt.imshow(sta[:,:,i],**kwargs)
14     plt.title(f"{num_timesteps - i} steps before the spike")
15     plt.axis('off')
16     plt.savefig(f"{num_timesteps - i}.png")
```

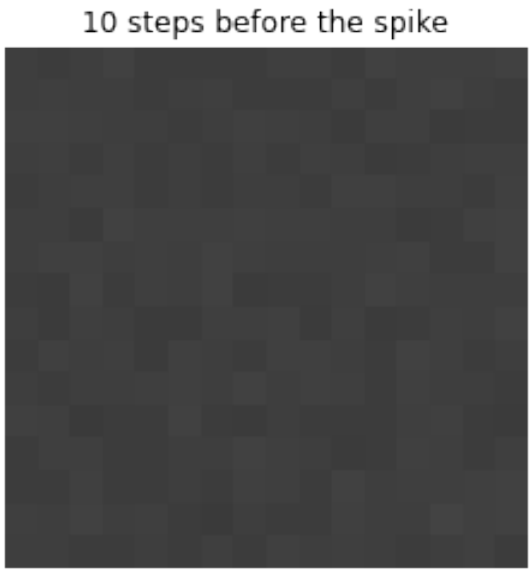

FIGURE 1. 10 steps before the spike

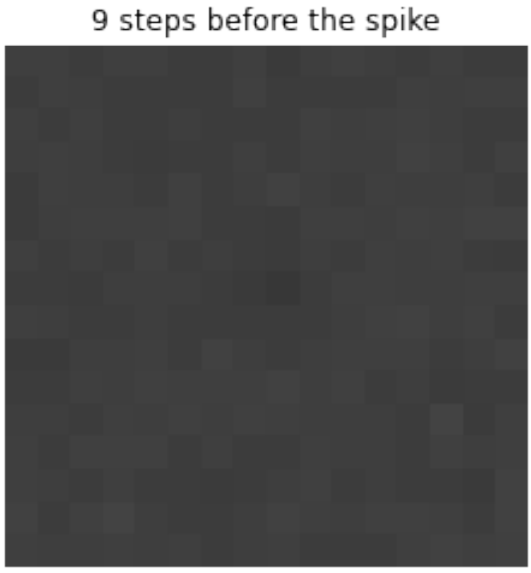

FIGURE 2. 9 steps before the spike

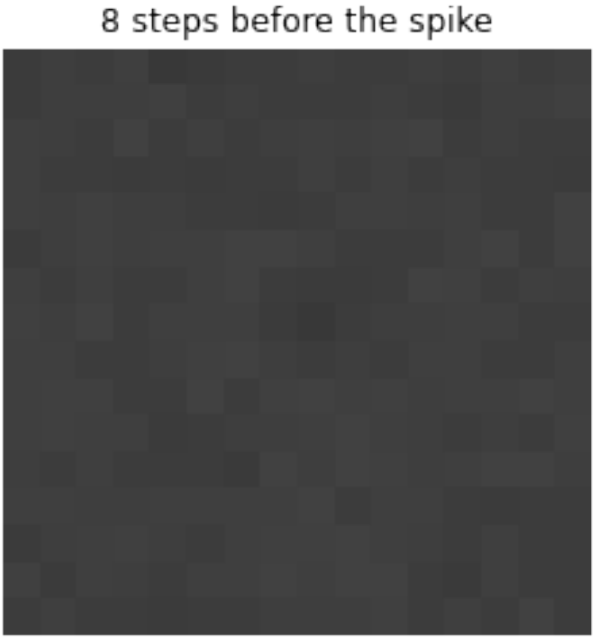

FIGURE 3. 8 steps before the spike

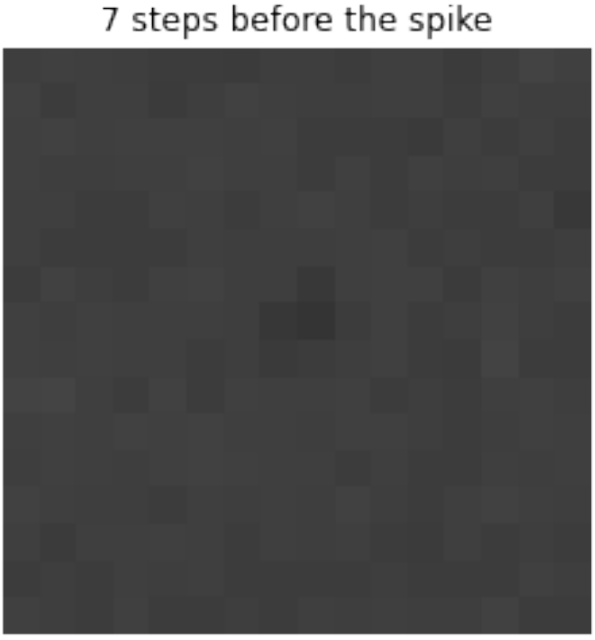

FIGURE 4. 7 steps before the spike

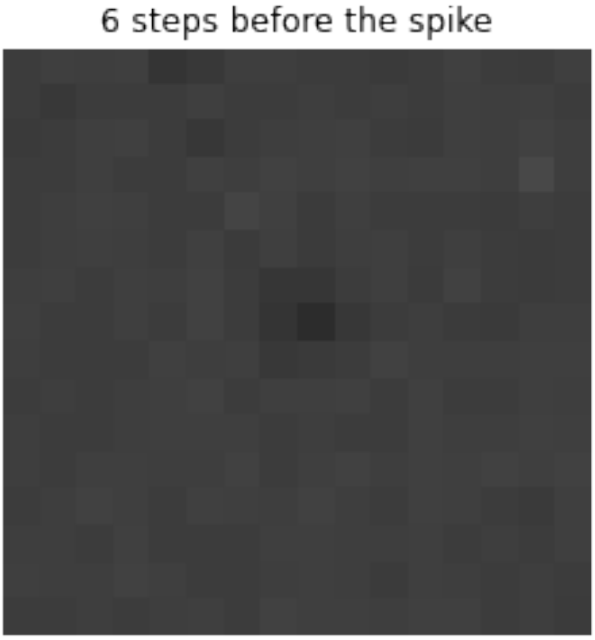

FIGURE 5. 6 steps before the spike

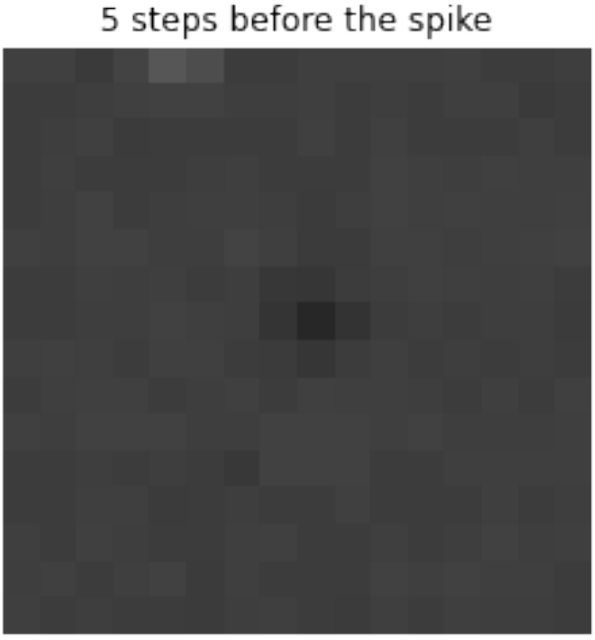

FIGURE 6. 5 steps before the spike

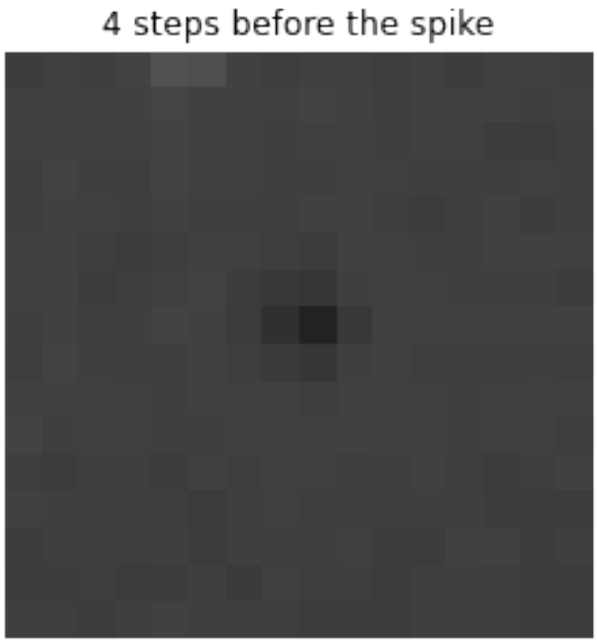

FIGURE 7. 4 steps before the spike

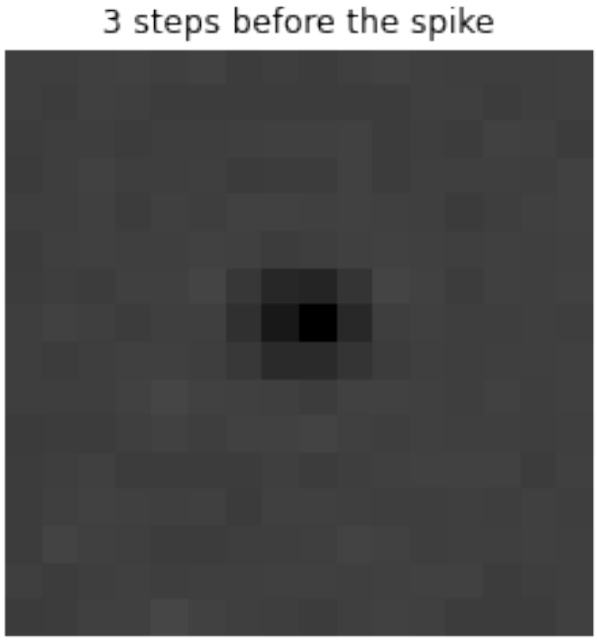

FIGURE 8. 3 steps before the spike

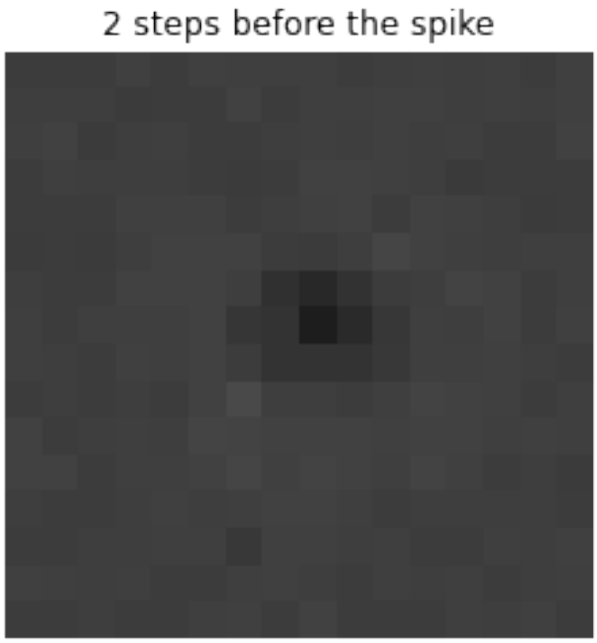

FIGURE 9. 2 steps before the spike

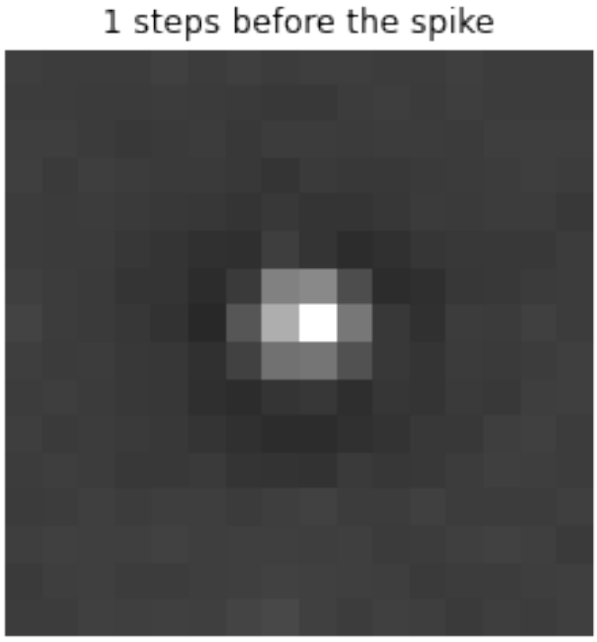

FIGURE 10. 1 steps before the spike

From the derived STA filter, one can conclude that as we get closer to spike, the center region of the STA is become darkener until the last step before the spike. Then, the flash burst in the last step before the spike so that we can observe a clear & distinct neural response. Hence, LGN cell is selective for gradual increase in darker regions until last step before spike and flash burst in last step before the spike under the assumption that STA model is linear receptive field. Additionally, by the analysis of stimulus and spike-train data, we can correlate the statistics of, i.e., mean and covariance, of the spatio-temporal distribution of stimuli corresponding to spike.

1.2. **Part B.** In the part b, we are asked to describe the changes in STA images across time by summing STA images over one of the spatial dimensions. Hence, we need to sum STA data in row and/or column wise. The following code snipped performs summation in both spatial dimensions and plot them side by side.

```
fig,axs = plt.subplots(1,2,figsize = (20,40))
axs[0].imshow(np.sum(sta,axis = 0),**kwargs)
axs[0].set_title(f"Row sum of STA")
axs[0].axis('off')
axs[1].imshow(np.sum(sta,axis = 1),**kwargs)
axs[1].set_title(f"Column sum of STA")
axs[1].axis('off')

plt.show()
```

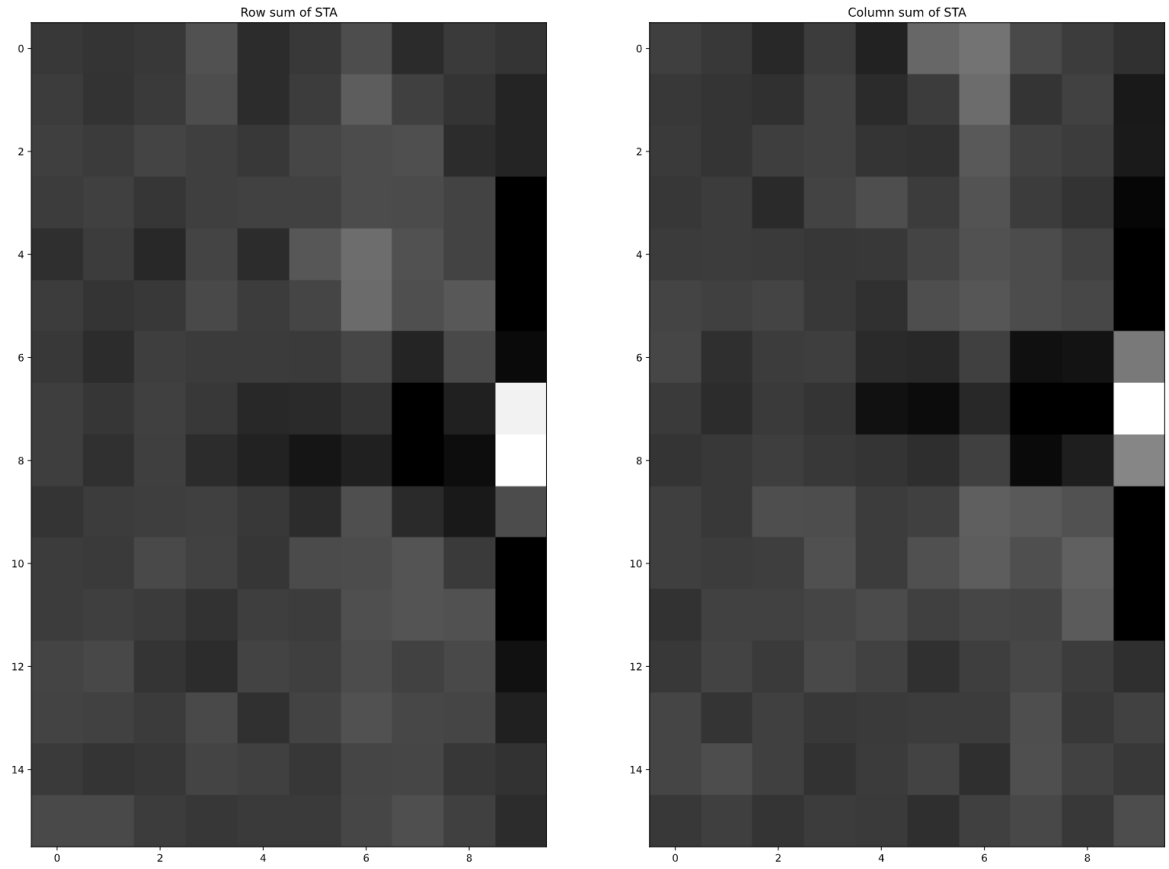

FIGURE 11. Summation across spatial dimension (column&row) wise respectively

The resulting images are in 16x10 format where absent 16 long spatial dimension is summed across the other spatial dimension. We can see from the above figure that row and column wise summation gives approximately symmetric distribution. As a clear distinction, as we get closer to left sides of the matrix (i.e., closer to spike regions) in time, comparatively, we can conclude that LGN cell is selective for a flash around pixel intensities 6-9. Additionally, one can conclude that the matrix is not space-time separable since the spikes solely depends on stimuli over time. Also, we can say that stimuli is not time-invariant (i.e., time-variant).

1.3. **Part C.** In this part of the question, we are asked to project the stimulus onto the STA image at a single time step prior to the spike and obtain the projection for each time sample by computing the Frobenius inner product between the stimulus image and the STA image.

Let's begin with the Frobenius inner product between the stimulus image and the STA image and how it performs. To be able compute, let's recall Frobenius inner product. Frobenius is basically inner product operation performs in matrix wise instead of vector wise. Note that Frobenius inner product is binary operation, i.e., returns a number, and denoted as follows

$$\langle A, B \rangle_F = \sum_{i,j} A_{ij} B_{ij} \text{ where A \& B must have same dimension} \tag{3}$$

Here is the computation of Frobenius inner product between the stimulus image and the STA image followed by normalization is placed.

```
project_stim = np.array([np.sum(sta[:,:, num_timesteps - 1] * _stim) for _stim
↪  in np.swapaxes(stim,0,2)])
project_stim /= project_stim.max()
```

Hence, we succesfully computed the normalized version of Frobenius inner product between the stimulus image and the STA image. Let's look at the shape of the histogram lonely as a intuitive way of understanding the distribution of the projections.

```
kwargs = dict(
    bins=100,
    alpha=.8,
    rwidth=.66,

)
plt.figure(figsize=(10, 5))
plt.hist(project_stim,color='g', **kwargs)
plt.title('Histogram of Stimilus Projections on STA')
plt.ylabel('Spike Frequency')
plt.xlabel('Normalized Stimulus Projection')
plt.show()
```

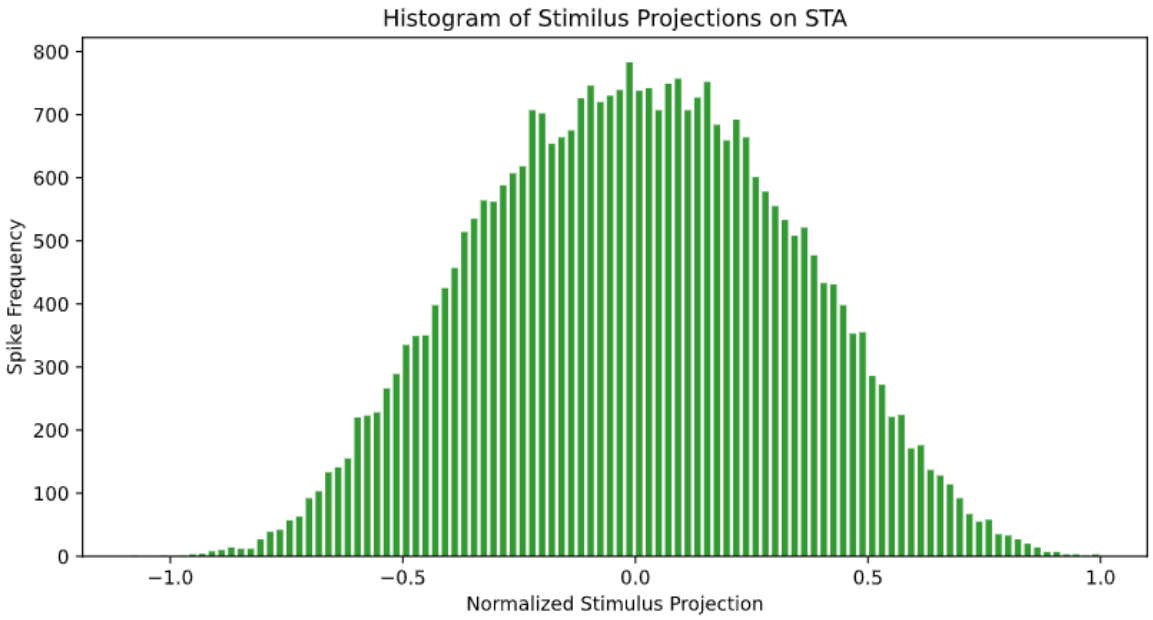

FIGURE 12. Histogram of normalized stimulus projections

Then, the next step is to create histogram for stimulus projections at time bins where a non-zero spike count was observed. To do that, I find the indexes of non-zero spikes followed by Frobenius inner product between the stimulus image and the STA image. Here is the code for computation of stimulus projections at time bins where a non-zero spike count was observed.

```
spike_times = counts.nonzero()[0]

project_stim_nonzero = np.array([np.sum(sta[:,:, num_timesteps - 1] *
↪  stim[:,:,i]) for i in spike_times])
project_stim_nonzero /= project_stim_nonzero.max()
```

As a prior step to plot both histograms on top of each other, let's look at the distribution of non-zero spikes normalized stimulus projections as follows.

```
plt.figure(figsize=(10, 5))
plt.hist(project_stim_nonzero, **kwargs,color='orange')
plt.title('Histogram of Stimilus Projections on STA for non-zero spikes')
plt.ylabel('Spike Frequency')
plt.xlabel('Normalized Stimulus Projection')
plt.show()
```

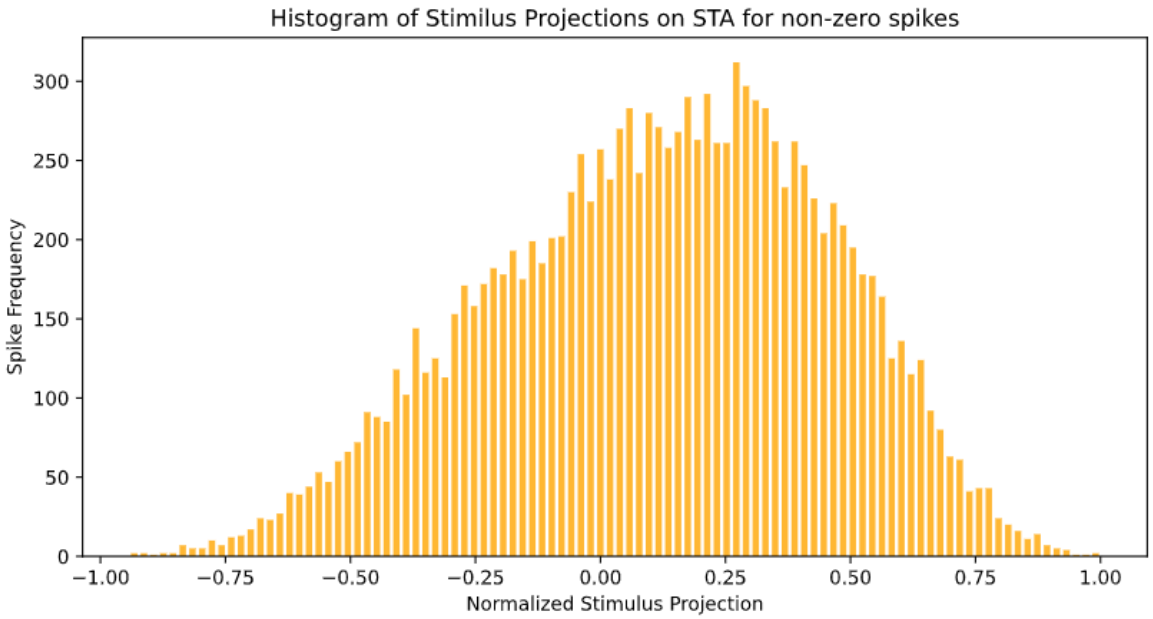

FIGURE 13. Histogram of Stimulus Projections on STA for non-zero spikes

Then, the final step is to plot both histograms on top of each other to see whether STA significantly discriminates spike-eliciting stimuli or not. So, let's plot both histograms on top of each other.

```
plt.figure(figsize=(10, 5))
plt.hist(project_stim, **kwargs)
plt.title('Comparison Histogram of Stimulus Projections on STA')
plt.hist(project_stim_nonzero, **kwargs)
plt.ylabel('Spike Frequency')
plt.xlabel('Normalized Stimulus Projection')
plt.legend(['For all Spikes','For nonzero spikes'])
plt.show()
```

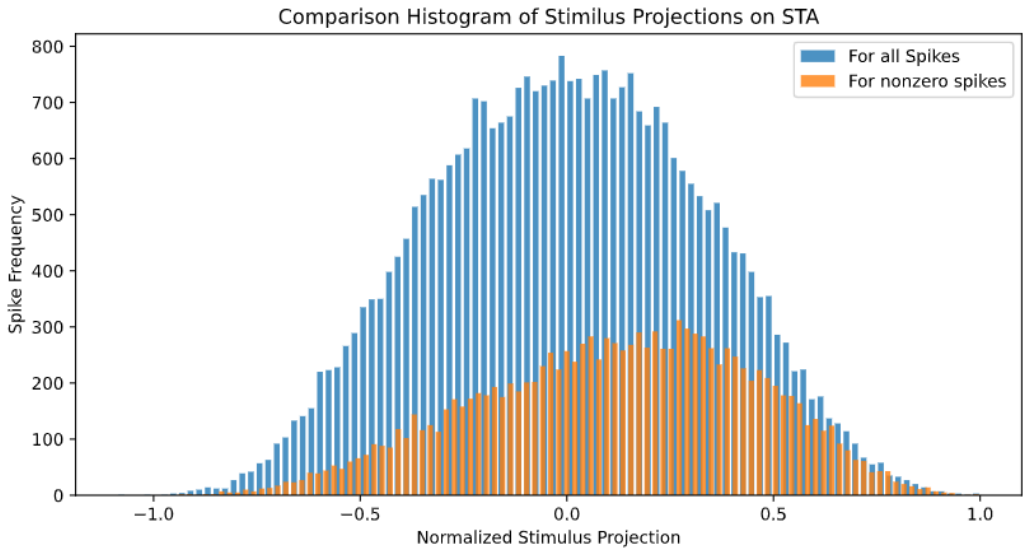

FIGURE 14. Comparison Histogram of Stimulus Projections on STA

As we can see from the FIGURE 14, the distribution of stimulus projections on STA images for all spikes are bringing to mind the centered (i.e., $\mu = 0$) Gaussian (Normal) distribution whereas the distribution of stimulus projections on STA images for non-zero spikes are reminding the Gaussian distribution where mean μ is in between $[0.2, 0.4]$. Since the shape of the distribution can fit into Gaussian line, we can say much about the both projections. Finally, strictly speaking, STA discriminates the spike-eliciting stimuli significantly since the projections of the stimulus onto the STA image with non-zero spikes turn centered Gaussian distribution into Gaussian with mean shifted version. Consequently, we can easily say that STA significantly discriminates spike-eliciting stimuli.

2. QUESTION 2

In this question, we are going to generate receptive fields and their responses of each neuron to given image **hw2_image.bmp** in a computational manner.

2.1. **Part A.** In this part, we are going to construct an on-center Difference-of-Gaussian's (DOG) center-surround receptive field centered at 0 as follows

$$D(x,y) = \frac{1}{2\pi\sigma_c^2} e^{(-x^2+y^2)/2\sigma_c^2} - \frac{1}{2\pi\sigma_s^2} e^{(-x^2+y^2)/2\sigma_s^2} \tag{4}$$

In the context of imaging, Difference of Gaussians (DoG) is a feature enhancement algorithm that involves the subtraction of one Gaussian blurred version of an original image from another, less blurred version of the original [2]. In the simple case of grayscale images, the blurred images are obtained by convolving the original grayscale images with Gaussian kernels having differing width (standard deviations)[2]. Hence, it is a classical 2-D image convolution operation (actually, it is cross-correlation) with DoG kernel. As a further information, blurring an image using a Gaussian kernel suppresses only high-frequency spatial information whereas subtracting one image from the other preserves spatial information that lies between the range of frequencies that are preserved in the two blurred images[2]. This is one of most significant property of DoG. Thus, the DoG is a spatial band-pass filter that attenuates frequencies in the original grayscale image that are far from the band center in 2-D domain. [2].

In our case, we will use the DoG as a 2-D filter to detect edges of source image. As we previously talked about the DoG applications on feature enhancement, it is powerful method to identify edges of the foreground objects. Furthermore, DoG filtering is very useful method in reduction of Gaussian noise that is generally active in higher spatial frequencies. Some classical filtering operations are sensitive to Gaussian noise in high spatial frequency so that they tend to enhance Gaussian noise while enhancing the features of foreground object. Hence, DoG is desirable algorithm when it comes to reducing the high frequency noises, it's the reason why DoG is preferable for vision problems with higher degree of Gaussian noise. So, let's generate 21x21 DoG receptive field with $\sigma_c = 2$ and $\sigma_s = 4$ and display in both 2-D & 3-D surface to understand the kernel in a visual way. Here is the Python code for generating DoG receptive field with given $\sigma_c = 2$, $\sigma_s = 4$ and size = (21x21).

```python
def DoG_Receptive_Field(std_c:int,std_s:int,size:int) -> np.ndarray:
    """
        Construct an on-center difference-of-gaussians (DoG) center-surround receptive field    centered at 0 with the given size. Generally, the Difference of Gaussian module is a   filter that  identifies edges.Additionally, DoG is a feature enhancement algorithm that involves the subtraction of one Gaussian blurred version of an original image from another, less blurred version of the original

            Arguments:
                - std_c (int) : Standard deviation of lower sigma
                - std_s (int) : Standard deviation of higher sigma
                - size  (int) : Size of DoG kernel

            Returns:
               - DoG_kernel (np.ndarray) : DoG kernel with given (size,size)

    """

    import math

    # Computing lower sigma kernel:
    low_sigma_kernel  = np.fromfunction(
        lambda x, y: (1/(2*math.pi*std_c**2)) * math.e ** ((-1*((x-(size-1)/2)**2+(y-(size-1)/2)**2))/(2*std_c**2)), (size, size)
        )

    # Computing higher sigma kernel:
    high_sigma_kernel = np.fromfunction(
        lambda x, y: (1/(2*math.pi*std_s**2)) * math.e ** ((-1*((x-(size-1)/2)**2+(y-(size-1)/2)**2))/(2*std_s**2)), (size, size)
        )

    # DoG kernel:
    DoG_kernel = low_sigma_kernel - high_sigma_kernel

    return DoG_kernel
```

Here, we successfully generate a function that gives DoG receptive field with any size and σ. Then, let's sample a 21x21 receptive field as follows

```python
DoG_kernel = DoG_Receptive_Field(**
dict(
    size = 21,
    std_c = 2,
    std_s = 4

))
```

Then, let's display the DoG kernel on both 2-D and 3-D surface as follows. Note that generic 2-D and 3-D filter plotting functions are written to be used in later parts of the question. The code for plotting filters will be only displayed below, and will be using just by calling the function when necessary. Hence, let's see the visualization of DoG filters. The first code snipped is for plotting the kernel on 2-D surface whereas the second one stands for 3-D filter visualization.

```
def plotFilter2D(kernel:np.ndarray,title:str,
                 cmap:str='coolwarm',
                 figsize:tuple = (10,10),
                 kwargs:dict = None) -> None:

    """
        Given the kernel, plot the kernel in 2-D surface.

            Arguments:
                - kernel (np.ndarray) : Kernel to be plotted
                - title  (str)        : Title of the figure
                - cmap   (str)        : Texture of the figure
                - figsize (tuple)     : Figure size
                - kwargs (dict)       : Additional arguments to plot if exists

            Returns:
                - None

    """

    plt.figure(figsize = figsize)

    if kwargs is not None:
        plt.imshow(kernel,cmap=cmap,**kwargs)
    else:
        plt.imshow(kernel,cmap=cmap)

    plt.title(title)
    plt.colorbar()
    plt.show()
```

Then, here is the code for 3-D filter visualization.

```
def plotFilter3D(kernel:np.ndarray,title:str,
                 kernel_name:str,
                 cmap:str='coolwarm',
                 figsize:tuple = (10,10),
                 kwargs:dict = None) -> None:

    """
        Given the kernel, plot the kernel in 2-D surface.

            Arguments:
                - kernel (np.ndarray) : Kernel to be plotted
                - title  (str)        : Title of the figure
                - kernel_name (str)   : The name of the filter/kernel
                - cmap   (str)        : Texture of the figure
                - figsize (tuple)     : Figure size
```

```
                - kwargs (dict)       : Additional orguments to plot if exists

            Returns:
                - None

    """

    fig_3D = plt.figure(figsize = (10,10))
    size = kernel.shape[0]
    ax_3D = plt.axes(projection='3d')
    x,y = np.meshgrid(
        np.arange( - int(size / 2), 1 + int(size / 2)),
        np.arange( - int(size / 2), 1 + int(size / 2))
                )

    if kwargs is not None:
        ax_3D.plot_surface(x, y, kernel,cmap = cmap, **kwargs)
    else:
        ax_3D.plot_surface(x, y,kernel, cmap = cmap)

    ax_3D.set_xlabel('x')
    ax_3D.set_ylabel('y')
    ax_3D.set_zlabel(f'{kernel_name}(x, y)')
    plt.title(title)
    plt.show()
```

Finally, we can plot the DoG filter with the parameters $\sigma_c = 2$, $\sigma_s = 4$ and size = (21x21).

```
kwargs = dict(
    rstride=1,
    cstride=1,
    edgecolor='none'
)

plotFilter2D(DoG_kernel,f'Difference of Gaussian (DoG) kernel with Stds {(2,4)}
↪  in 21x21 form')

plotFilter3D(DoG_kernel,f'Difference of Gaussian (DoG) kernel with Stds {(2,4)}
↪  in 21x21 form','DoG', kwargs = kwargs)
```

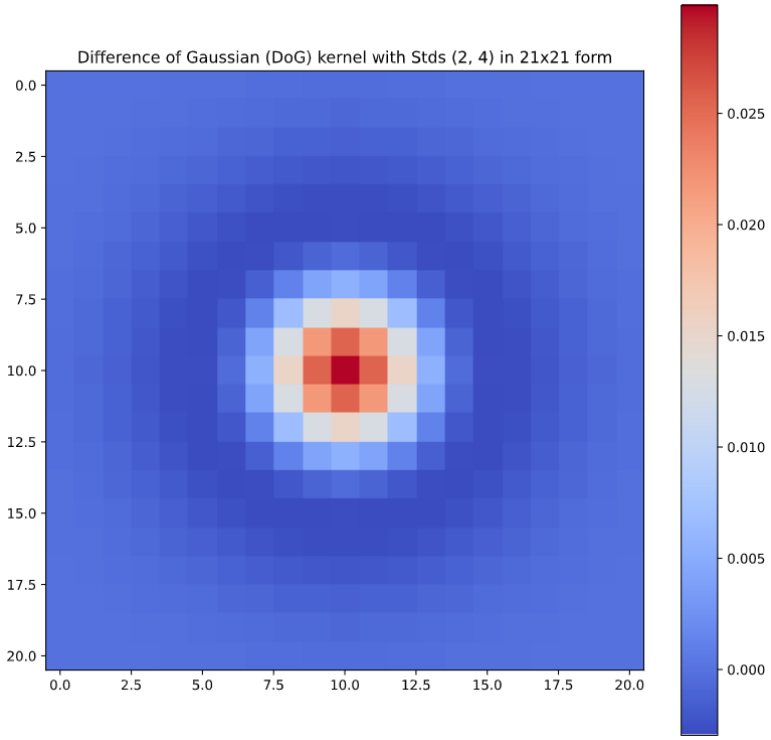

FIGURE 15. Difference of Gaussian (DoG) kernel with Stds (2,4) in 21x21 form

3-D visualization of Gaussian based kernels are better way to get insights from the distributions. So, here I plotted our 21x21 DoG kernel in 3-D canvas.

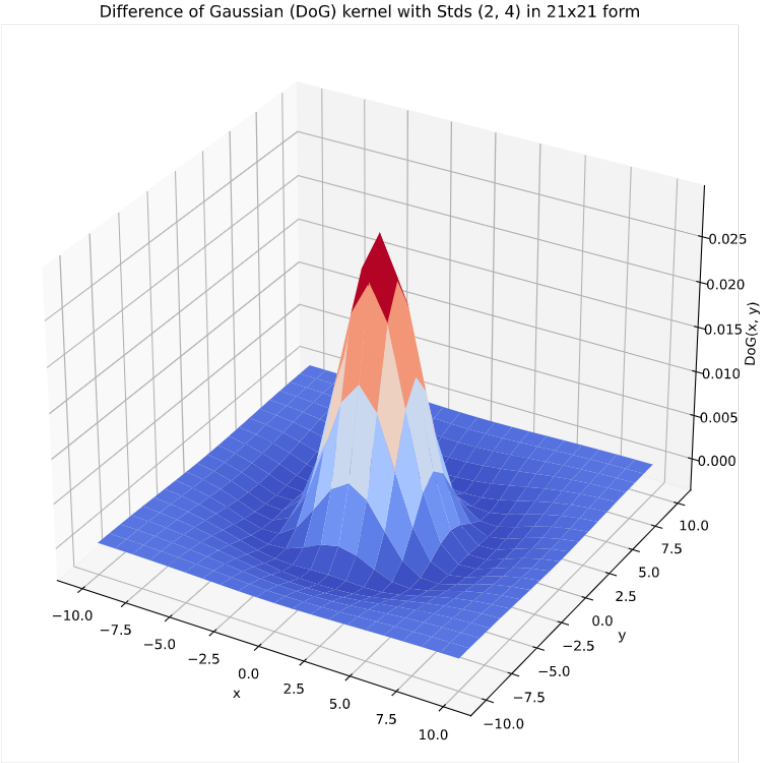

FIGURE 16. Difference of Gaussian (DoG) kernel with Stds (2,4) in 21x21 form

2.2. **Part B.** In this part of the question, we are assuming that there is a separate LGN neuron with a receptive field centered on each pixel in the image with the knowledge that neurons in lateral geniculate nuclei (LGN) have DOG receptive fields. So, here we are going to compute the responses of each neuron to the image given in **hw2_image.bmp**.

As a prior step along the computation of responses of each neuron to given image, let's visualize the given image as follows.

```
image = plt.imread('hw2_image.bmp')
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.axis('off')
plt.show()
```

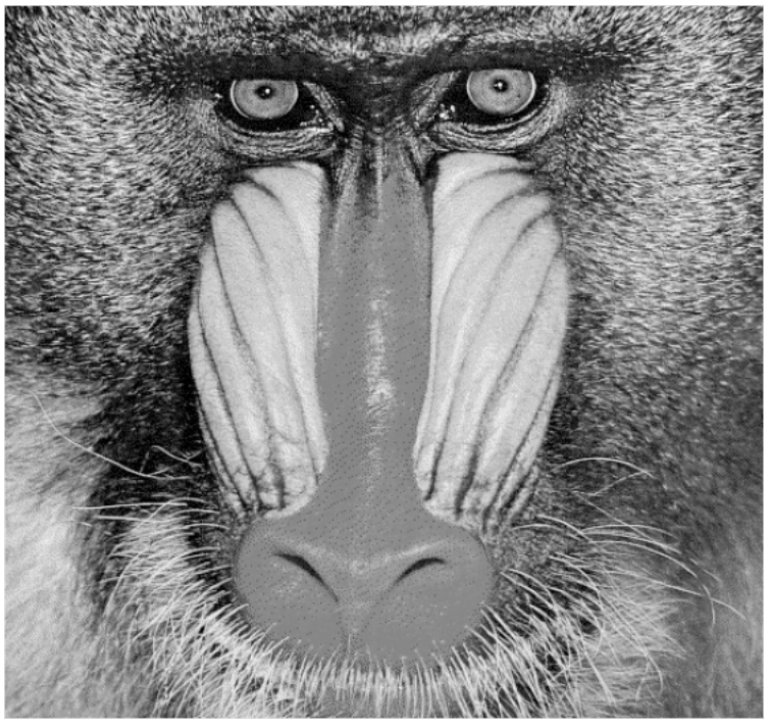

FIGURE 17. Given input image

To be able to compute the responses of each neuron to the image given in **hw2_image.bmp**, we need to create sliding window over the image such DoG receptive field kernel can moves along the spatial dimensions of the input image. In the context of computational neuroscience, the activity of an LGN cell can be explained as the amount of overlap between DoG receptive field and its input. In mathematics domain, this process is called **convolution**. 2-D Convolution is performed by multiplying and accumulating the instantaneous values of the overlapping samples corresponding to input images and its filter. So, let's create a convolution function as follows. But, as a prior step to create convolution function, we need to have sliding window over the padded image (Zero padding for **'same'** convolution operation, i.e., pad the image so that resulting image have the same spatial dimensions as before). Let's see the Python code for sliding window.

```python
class _Window(object):
  """
    This class creates a generator that slide over the given image with
    predefined step and window size.

      Attributes:
        - image      : input image to be sliding
        - step_size  : it determines how much slice the generator extract
        - dims       : resulting image's dimensions

      Methods:
        - __iter__   : creates a generator over the image with given patch
  ↪  dimensions

  """
  def __init__(self,image : np.ndarray, step_size:tuple, dims: tuple):
    """
      Creates an constructor for _Window class, this method encapsulates
      all necessary data to generate windows over image.

    """
    self.image = image
    self.dims = dims
    self.step_size = step_size

  def __iter__(self):
    """ Generator function to window over image. """
    for x in range(self.dims[0]):
      for y in range(self.dims[1]):
        yield self.image[x:x + self.step_size[0], y:y + self.step_size[1]]
```

Hence, all we need to do is to pad the image with zeros if necessary, then multiply the corresponding entries of image patches with DoG kernel, then accumulate the resulting multiplication to see fill the corresponding entry of convolved image. Here is the Python code for convolution.

```python
def Conv2D(source_image : np.ndarray, kernel: np.ndarray) -> np.ndarray:
  """
   Convolution is the process of adding each element of the image to its local
  ↪  neighbors,weighted by the kernel. This is related to a form of mathematical
  ↪  convolution.

      Arguments:
        - source_image   (np.ndarray) : Gray scale image to be convolved
        - kernel         (np.ndarray) : Kernel to be sliding over the image

      Returns:
        - conv_image     (np.ndarray) : Resulting convolved image
  """

  assert (len(source_image.shape)   == 2), "Image is not gray scale"
  assert (len(kernel.shape) == 2), "Kernel is not in required size"

  source_image =  np.asarray(source_image).clip(0,255).copy()
```

```
    H,W = source_image.shape
    k_h,k_w = kernel.shape
    padded_image = np.pad(array = source_image, pad_width = max(k_w,k_w) // 2 + 1,
    ↪  mode = 'constant')

    new_H,new_W = padded_image.shape
    h_pad , w_pad  =  (new_H - H) , (new_W - W)

    # Creating sliding window:
    image_window = _Window(padded_image,(k_h,k_w),(new_H - h_pad, new_W - w_pad))

    #kernel = np.flipud(np.fliplr(kernel))
    # Main operation:
    conv_image = [(patch * kernel).sum() for patch in image_window]

    return np.array(conv_image).reshape(H,W)
```

Finally, we can compute the responses of each neuron to the given image. Note that even the given image has 3 channels, it is actually replica of 1 channel so that any random channel can be slicing as a source image for the following computations.

```
gray_image = image[:,:,0]
filtered_image = Conv2D(gray_image,DoG_kernel)

plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(filtered_image),cmap = 'gray')
plt.title('Convolved image by DoG kernel')
plt.axis('off')
plt.show()
```

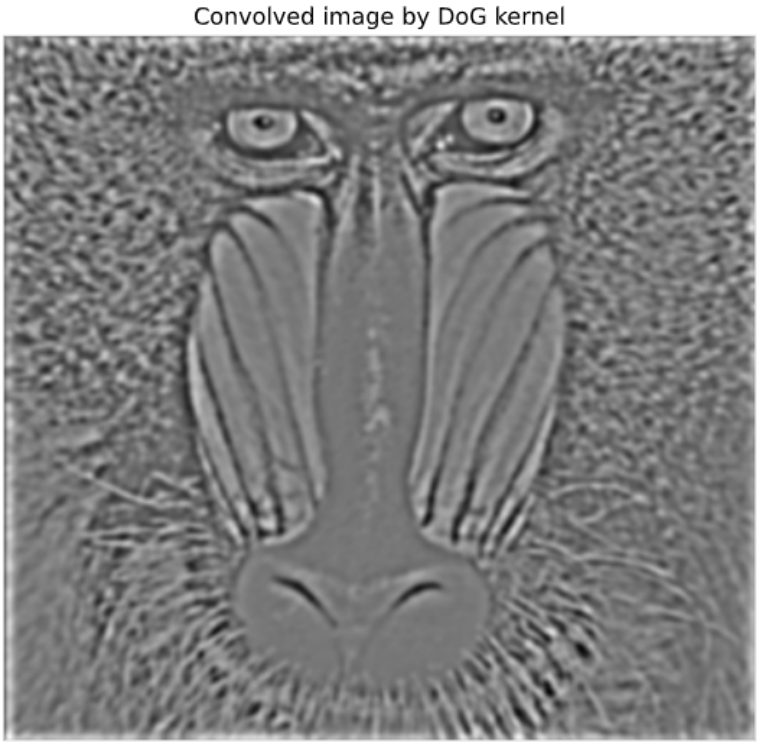

FIGURE 18. Convolved image with DoG receptive field.

Note that MIN-MAX scaling operation is applied on resulting image to plot in matplotlib's input format, i.e., the pixel intensities should be in the range 0-1 or 0-255. It has no functionality other than that in our case. Just be concrete on the discussion, let's provide the code for MIN-MAX scaler.

```
def min_max_scaler(matrix : np.ndarray)-> np.ndarray :
  """
    Given the matrix, apply normalization as follow:

        norm_matrix = (matrix - min_val) / (max_val - min_val)

    Arguments:
      -  matrix     (np.ndarray) : Input Matrix

    Returns:
      - norm_matrix (np.ndarray) : Normalized version of matr

  """
  matrix = np.asarray(matrix).copy()
  max_val = matrix.max()
  min_val = matrix.min()

  return (matrix - min_val) / (max_val - min_val)
```

2.3. **Part C.** In this part of the question, we are asked to build an edge detector by thresholding the neural activity image (i.e., setting all values above a certain threshold to 1 and the remainder to 0.). Hence, we need to find a optimal threshold value so that we can identifies edges in distinct way. To do that, one can approach different methods such as manual thresholding, i.e., trial-and-error fashion, or we can use automatic thresholding algorithms such as Otsu method. Otsu's automatic thresholding is a method that separates background and foreground of image with automatically computed threshold value. This threshold is computed via minimizing the within-class variance or maximizing the inter-class variability that yields same result. However, note that computing threshold value by maximizing the inter-class variance is more computationally efficient method. Hence, I created a Otsu thresholding method for finding automatic threshold value of the given source image. Here is the code for Otsu thresholding.

```
def otsu_threshold(source_image: np.ndarray) -> np.ndarray:
    """
    Otsu's automatic thresholding method that seperates background and
↪   foreground of image with automatically computed threshold value. This
↪   threshold is computed via minimizing the within-class variance or maximizing
↪   the inter-class variability that yields same results. However, note that
↪   computed threshold value by maximizing the inter-class variance is more
↪   computationally efficient method.

      Arguments:
        - source_image (np.ndarray) : Source image to be thresholded by Otsu

      Returns:
        - thres_image  (np.ndarray) : Resulting image after applying Otsu's
↪   method

  """
```

```
12
13      assert (len(source_image.shape) == 2), "Image is not gray scale"
14
15      source_image =  np.asarray(source_image).clip(0,255).copy()
16
17      hist =  [np.sum(pixel == source_image) for pixel in np.arange(256)]
18
19
20      otsu_thres = 0
21      var = 0
22      max_val = 256
23
24      for thres in np.arange(256):
25
26        # For first class, mean and variance:
27        w_0 = sum(pixel for pixel in hist[:thres])
28        mu_0 = sum([i * hist[i] for i in range(thres)]) / w_0 if w_0 > 0 else 0
29
30        # For second class, mean and variance:
31        w_1 = sum(pixel for pixel in hist[thres:])
32        mu_1 = sum([j * hist[j] for j in range(thres, max_val)]) / w_1 if w_1 > 0
          ↪  else 0
33
34        # calculating inter-class variance
35        s = w_0 * w_1 * (mu_0 - mu_1) ** 2
36
37        # Set threshold if new inter class variance is bigger than previous
38        otsu_thres = thres - 1 if s > var else otsu_thres
39        var = s if s > var else var
40
41      #print(f'Self written Otsu threshold value is {otsu_thres}')
42
43      # Apply thresholding: Hence
44      source_image[source_image >= otsu_thres] = 255
45      source_image[source_image < otsu_thres] = 0
46
47      return source_image.astype(np.uint8)
```

As we discuss, Otsu thresholding is automatic way of thresholding that is desirable for autonomous tasks. Additionally, Otsu thresholding performs good if calculated histogram comes from the bimodal distribution (having two different modes, clear & distinctive local peaks, etc.). Eventually, Otsu method will find proper value of thresholding that successfully separates background from foreground if tonal distribution of the image comes from the bimodal distribution. Hence, performance of Otsu method is limited. Therefore, it is not always good idea to apply Otsu thresholding before having a idea of tonal distribution of images. In a nutshell, it is always good idea to tune the threshold value in hand-crafted way so I also create a function that separates the background from the foreground in manual way.

```python
def manuel_thresholding(source_image: np.ndarray,threshold:int) -> np.ndarray:
    """
        Given the source image and threshold, apply thresholding (i.e., setting all values above a certain threshold to 1 and the remainder to 0.

            Arguments:
                - source_image (np.ndarray) : Source image to be thresholded
                - threshold    (int)        : Threshold value

            Returns:
                - thres_image  (np.ndarray) : Resulting image after applying threshold

    """


    assert (len(source_image.shape) == 2), "Image is not gray scale"

    source_image =  np.asarray(source_image).clip(0,255).copy()

    # Apply thresholding:
    source_image[source_image >= threshold] = 255
    source_image[source_image < threshold] = 0

    return source_image.astype(np.uint8)
```

Now, we can see the edge detector's performance so let's plot the resulting images after both manual and Otsu thresholding method. Note that I applied MIN-MAX scaling operation on filtered images since the output of the convolution operation results the pixel intensity values in range $[-50, 50]$. By doing that, I transformed the pixel values into range $[0, 255]$ as 8-bit integers. So, we can pass the resulting image into both Otsu and manual thresholding to see the edge detector's performance.

```python
img = np.array(min_max_scaler(filtered_image) * 255, dtype = np.uint8)
plt.figure(figsize=(7,7))
plt.imshow(otsu_threshold(img),cmap = 'gray')
plt.title('Convolved image by DoG kernel')
plt.axis('off')
plt.show()

plt.figure(figsize=(7,7))
plt.imshow(manuel_thresholding(img,115),cmap = 'gray')
plt.title('Convolved image by DoG kernel')
plt.axis('off')
plt.show()
```

FIGURE 19. Resulting image after manual thresholding

FIGURE 20. Resulting image after otsu thresholding

As we can see from the figures, we successfully identifies the edges of the foreground object. In manual thresholding case with the threshold value 115, we detect smoother edges compared to Otsu case. In otsu binarization, we see that even we identifies a edges of the object, the contour of the foreground is also visible so that manual thresholding gives better edge performance in this case. Additionally, we can say that Otsu's binarization is approximately dilated version of the manual thresholding.

2.4. **Part D.** In this part, as similar to part A, we are asked to construct receptive field with 21x21 form but in this case, the distribution of the receptive field comes from the Gabor. Mathematically, we can represent the Gabor kernel as follows

$$D(\vec{x}) = \exp\left(-(\vec{k}(\theta)\cdot\vec{x})^2/2\sigma_l^2 - (\vec{k_\perp}(\theta)\cdot\vec{x})^2/2\sigma_w^2\right)\cos\left(2\pi\frac{\vec{k_\perp}(\theta)\cdot\vec{x}}{\lambda}+\phi\right) \tag{5}$$

Here $\vec{k_\perp}(\theta)$ and θ is a unit vector with the orientation θ, $\vec{k_\perp}(\theta)$ is a unit vector orthogonal to $\vec{k_\perp}(\theta)$ and $\theta, \sigma_l, \sigma_w, \lambda$ and ϕ are parameters of the Gabor kernel. As provided, we have $\theta = \pi/2$, $\sigma_l = \sigma_w = 3$, $\lambda = 6$, and $\phi = 0$. So, let's begin with the discussion of the Gabor filter and its applications to our case. Gabor is a linear filter used for texture analysis, edge detection, feature extraction, etc. which essentially means that it analyzes whether there is any specific frequency content in the image in specific directions in a localized region around the point or region of analysis[3]. Frequency and orientation representations of Gabor filters are claimed by many contemporary vision scientists to be similar to those of the human visual system [3] as we are computing responses of each neuron to the given image. Hence, the visual cortex of some mammals can be expressed by these filters. As we discuss, Gabor filter considers the optimal localization properties in both spatial & frequency domain with orientations so that they are useful in egde detection, segmentation problems. We can consider Gabor filters as a sinusoidal oriented 2-D Gaussian filters that considers the spatial neighborhoods in cross-correlations. Let's see a figure of Gabor filter as a intuitive way of understanding the internal structure.

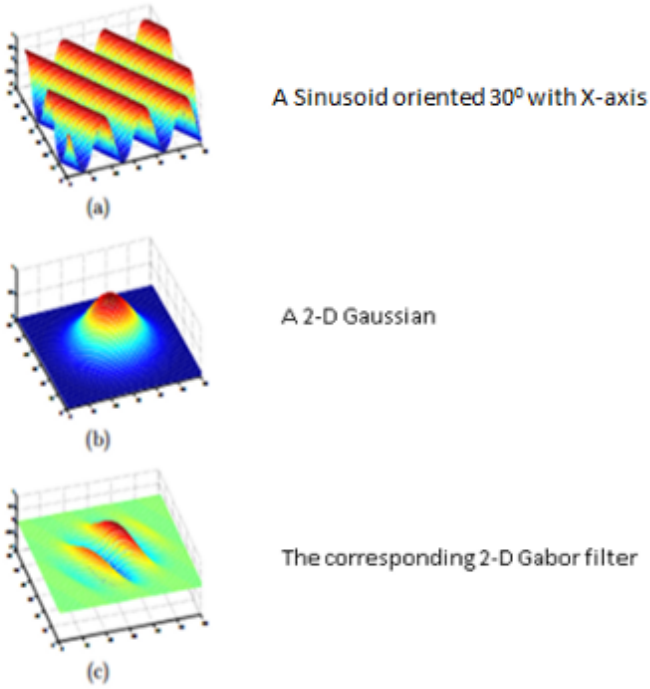

FIGURE 21. A 2-D Gabor filter obtained by modulating the sine wave with a Gaussian

As we can see from the figure, sinusoidal component of Gabor filters is spatially localized. Intuitively, we can use different orientation of 2-D Gabor filters to extract spatially meaningful features from the images. Then, we can move to construction of Gabor receptive fields as follows.

```
def Gabor_Receptive_Field(std_l:int,std_w:int,
                          theta:float,Lambda:int,
                          psi:int, size:tuple) -> np.ndarray:
    """
        Gabor Receptive Field is a linear filter used for texture analysis,
    ↪  which essentially            means that it analyzes whether there is any
    ↪  specific frequency content in the image in           specific directions in
    ↪  a localized region around the point or region of analysis

            Arguments:
                - std_l  (int)   : Standard deviation of lower sigma
                - std_w  (int)   : Standard deviation of higher sigma
                - theta  (float) : Orientation
                - Lambda (int)   : Gabor filter parameter
                - psi    (int)   : Gabor filter parameter
                - size   (tuple) : Size of Gabor kernel

            Returns:
               - Gabor_kernel (np.ndarray) : Gabor kernel with given (size,size)

    """


    import math

    x, y = np.meshgrid(np.arange(- int(size[0] / 2), 1 + int(size[0] / 2)),
                       np.arange(- int(size[1] / 2), 1 + int(size[1] / 2)))


    # Rotation
    x_theta = x * np.cos(theta) + y * np.sin(theta)
    y_theta = -x * np.sin(theta) + y * np.cos(theta)

    # Main function for creating Gabor filter:
    Gabor_kernel = np.exp(-.5 * (x_theta ** 2 / std_l ** 2 + y_theta ** 2 /
    ↪  std_w ** 2)) * np.cos(2 * np.pi / Lambda * x_theta + psi)


    return Gabor_kernel
```

As provided, we have $\theta = \pi/2$, $\sigma_l = \sigma_w = 3$, $\lambda = 6$, and $\phi = 0$ in this part so let's get a sample from Gabor kernel with 90° orientation as follows.

```
theta_90 = np.pi / 2
size = (21,21)
kwargs_kernel = dict(std_l = 3,std_w = 3,
                     psi = 0,Lambda = 6,
                     size = size)

Gabor_90_kernel = Gabor_Receptive_Field(theta = theta_90,**kwargs_kernel)
```

Finally, let's see the visualization of Gabor filter in both 2-D and 3-D surfaces as follows.

```
kwargs = dict(
    rstride=1,
    cstride=1,
    edgecolor='none'

)

plotFilter2D(Gabor_90_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,90,size}'
)



plotFilter3D(Gabor_90_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,90,size}',kernel_name = 'Gabor',kwargs=kwargs)
```

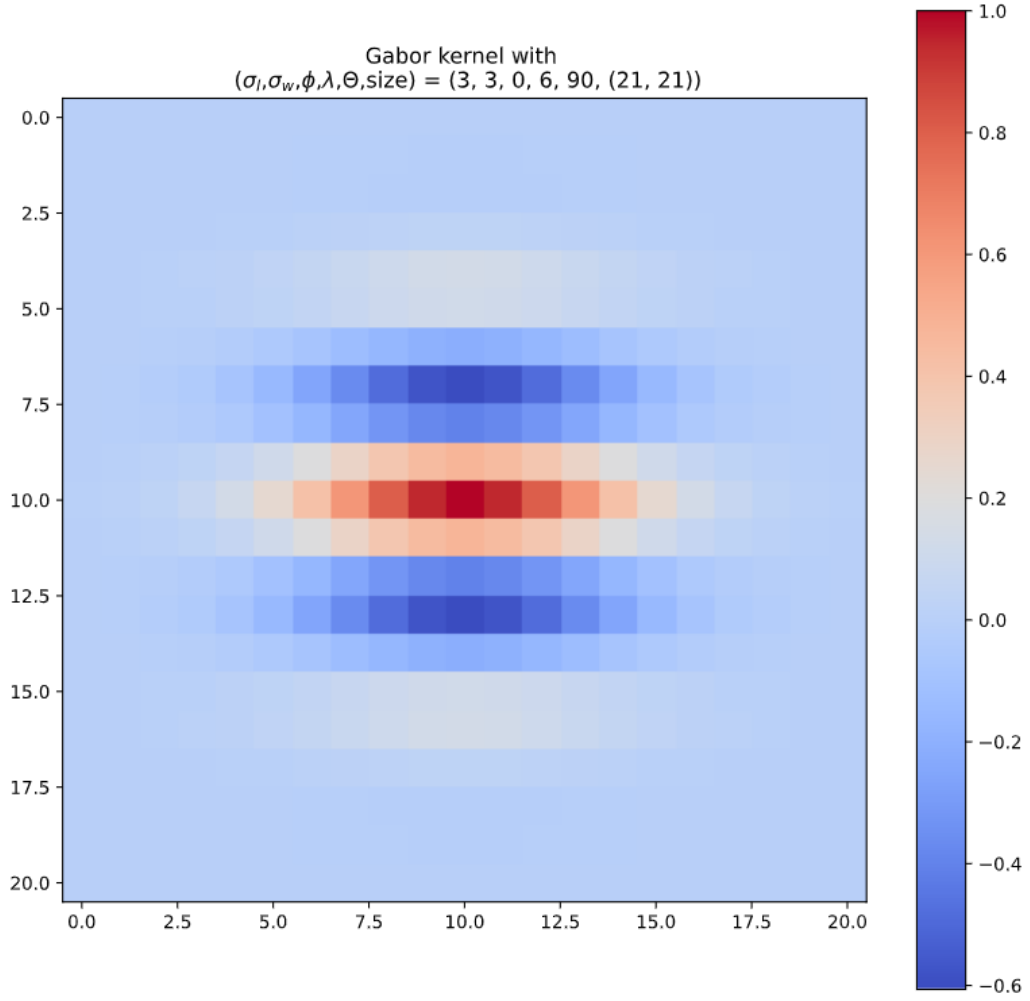

FIGURE 22. Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,90,(21,21))$

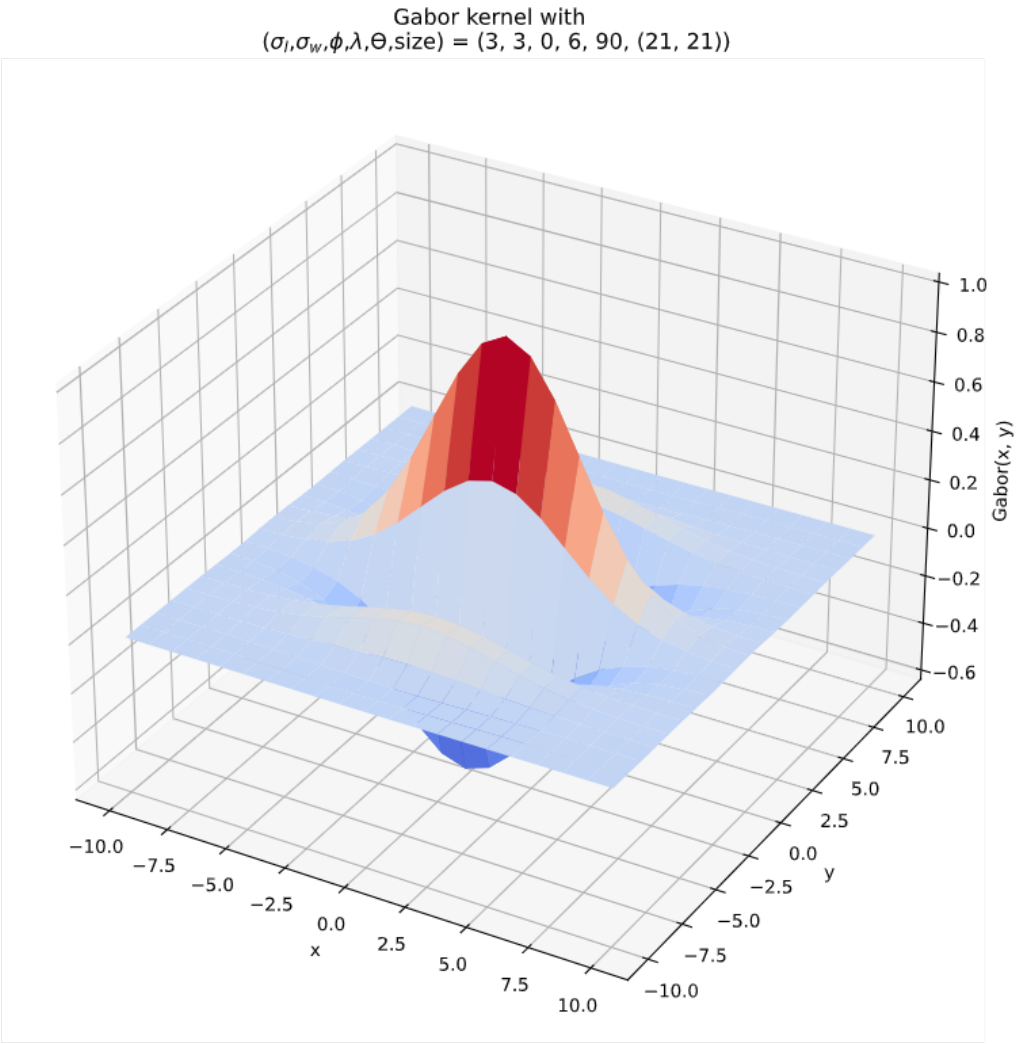

FIGURE 23. Gabor kernel with (σ_l,σ_w,ϕ,λ,θ,size) = (3,3,0,6,90,(21,21))

2.5. **Part E.** In this part of the question, assume that there is a separate V1 neuron with a receptive field centered on each pixel in the image with the knowledge that simple cells in V1 have Gabor receptive fields. We need to compute the responses of each neuron to the image given in hw2_image.bmp. Finally, we are asked to discuss the function of Gabor filter. Note that this discussion is already done in previous part in detail. But, here just for the sake of simplicity, I made the similar discussion of Gabor filter. Gabor is a linear filter used for texture analysis, edge detection, feature extraction, etc. which essentially means that it analyzes whether there is any specific frequency content in the image in specific directions in a localized region around the point or region of analysis[3]. Frequency and orientation representations of Gabor filters are claimed by many contemporary vision scientists to be similar to those of the human visual system [3] as we are computing responses of each neuron to the given image . Hence, the visual cortex of some mammals can be expressed by these filters as we can see in the literature computational neuroscience. As we discuss, Gabor filter considers the optimal localization properties in both spatial & frequency domain with orientations so that they are useful in egde detection, segmentation problems. We can consider Gabor filters as a sinusoidal oriented 2-D Gaussian filters that considers the spatial neighborhoods in cross-correlations. In a nutshell, we can think Gabor filter as linear filter such that optimal localization properties in both spatial & frequency domain with orientations is preserved so that they become

sinusoidal Gaussian envelopes. Then, let's move to computing the responses of each neuron to the image given via 2-D convolution operation as we already implemented in previous parts.

```
Gabor_result_1 = Conv2D(image[:,:,0],Gabor_90_kernel)

plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_1),cmap = 'gray')
plt.title('Convolved image by DoG kernel')
plt.axis('off')
plt.show()
```

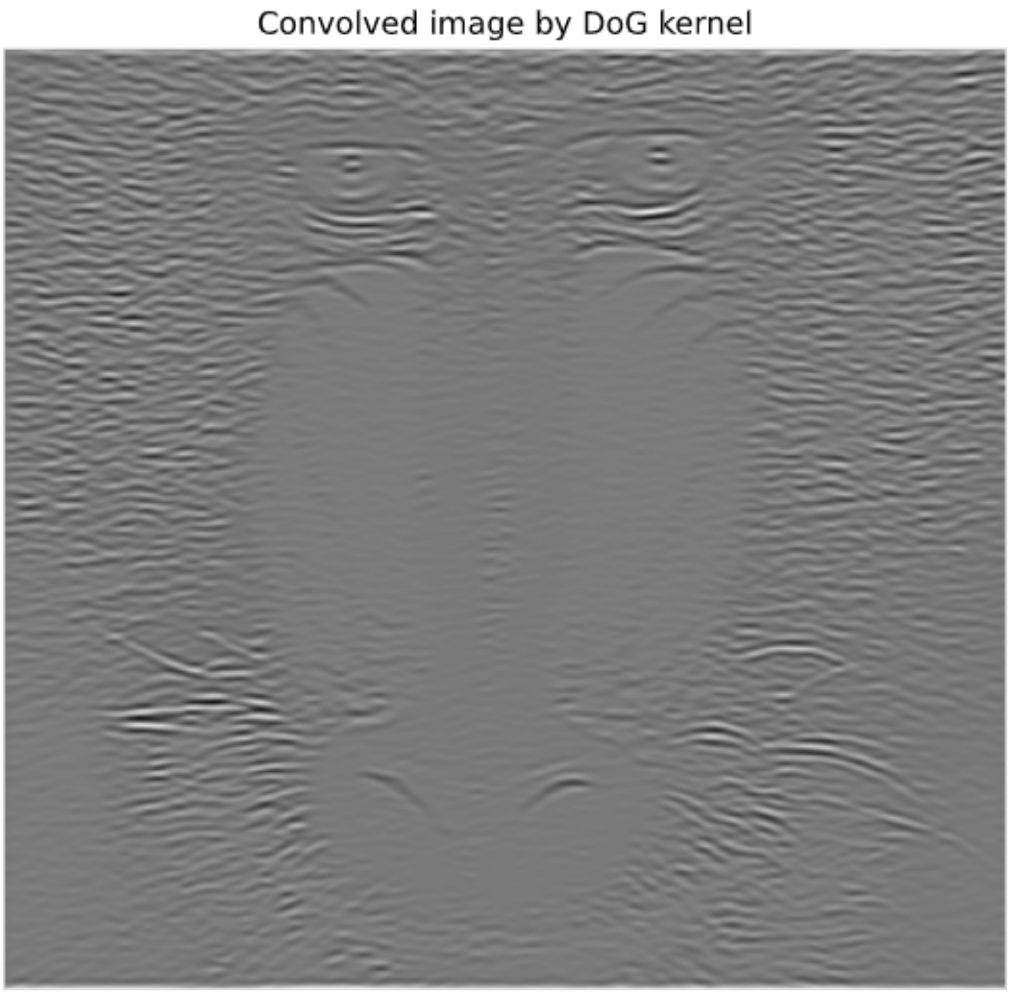

FIGURE 24. Convolved image by Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,90,(21,21))$

2.6. **Part F.** In this part of the question, we are asked to Construct 4 Gabors with $\theta = 0, \pi/2, \pi/3, \pi/2$, then compute combined neural responses to the image hw2_image.bmp, by summing the outputs of the individual receptive fields (for different θ). As we previously discuss the usage of Gabor filters and its applications in digital image domain, they are generally more than 1 Gabor filters with different orientation angles, and they are used together (by summing) to boost the performance of edge detection, texture analysis, etc as it is very common application of Gabor filter. Hence, let's construct Gabor receptive field, visualize them in both 2-D and 3-D surface then compute the individual neural responses to given image. Finally, then, we can sum the all computed neural responses to get end result. Here is the code & visualization for For $\theta = 0$:

```
theta_0 = 0
Gabor_0_kernel = Gabor_Receptive_Field(theta  = theta_0, **kwargs_kernel)

plotFilter2D(Gabor_0_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,theta_0,size}')

plotFilter3D(Gabor_0_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,theta_0,size}',kernel_name = 'Gabor',kwargs=kwargs)

Gabor_result_2 = Conv2D(image[:,:,0],Gabor_0_kernel)
plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_2),cmap = 'gray')
plt.title(f'Convolved image by DoG kernel with  \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,0,size}')
plt.axis('off')
plt.show()
```

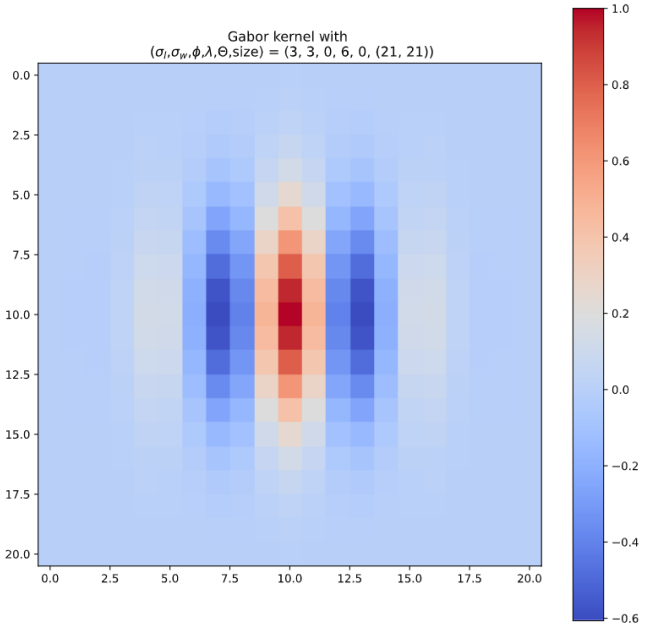

FIGURE 25. Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,0,(21,21))$

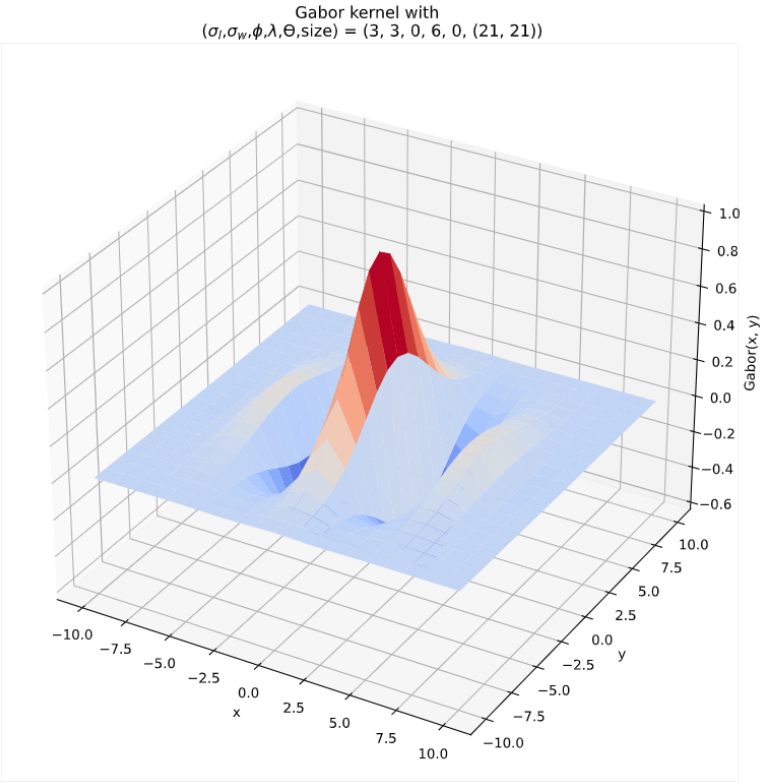

FIGURE 26. Gabor kernel with (σ_l,σ_w,ϕ,λ,θ,size) = (3,3,0,6,0,(21,21))

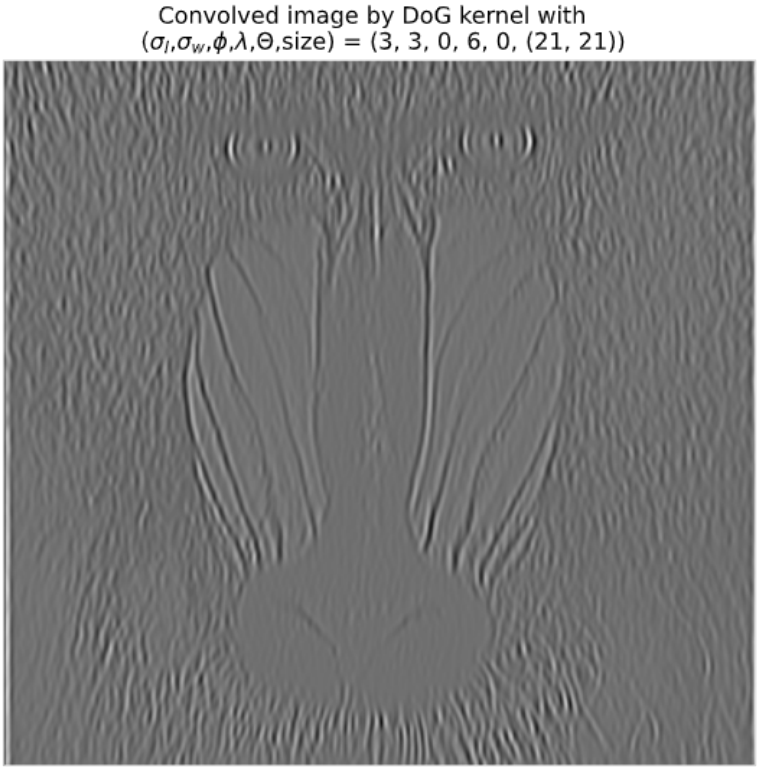

FIGURE 27. Convolved image with Gabor kernel with (σ_l,σ_w,ϕ,λ,θ,size) = (3,3,0,6,0,(21,21))

Here is the code & visualization for $\theta = \pi/6$.

```
theta_30 = np.pi / 6

Gabor_30_kernel = Gabor_Receptive_Field(theta = theta_30,**kwargs_kernel)
plotFilter2D(Gabor_30_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,30,size}')

plotFilter3D(Gabor_30_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,30,size}',kernel_name = 'Gabor',kwargs=kwargs)


Gabor_result_3 = Conv2D(image[:,:,0],Gabor_30_kernel)


plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_3),cmap = 'gray')
plt.title(f'Convolved image by DoG kernel with  \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,30,size}')
plt.axis('off')
plt.show()
```

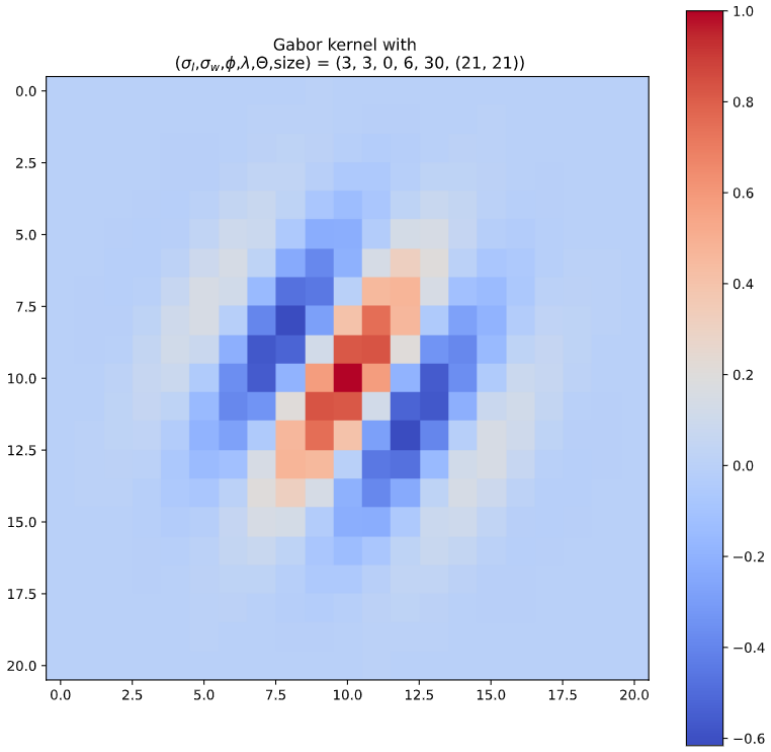

FIGURE 28. Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,30,(21,21))$

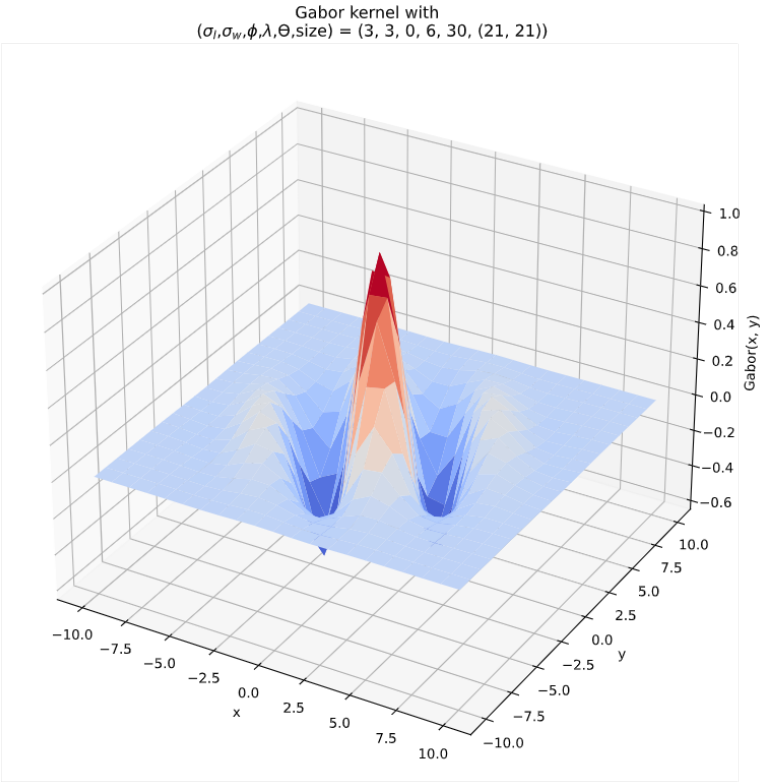

FIGURE 29. Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,30,(21,21))$

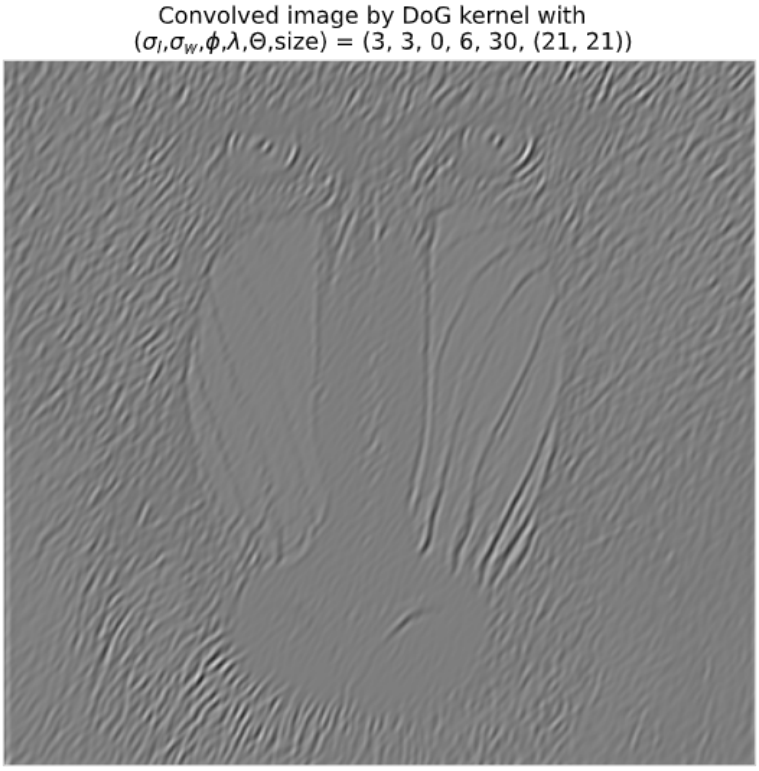

FIGURE 30. Convolved image with Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,30,(21,21))$

Here is the code & visualization for $\theta = \pi/3$.

```
theta_60 = np.pi / 3

Gabor_60_kernel = Gabor_Receptive_Field(theta = theta_60,**kwargs_kernel)

plotFilter2D(Gabor_60_kernel,
title = f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\theta$,size) =
↪  {std_l,std_w,phi,Lambda,60,size}'

)

plotFilter3D(Gabor_60_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\theta$,size) =
↪  {std_l,std_w,phi,Lambda,60,size}',kernel_name = 'Gabor',kwargs=kwargs)



Gabor_result_4 = Conv2D(image[:,:,0],Gabor_60_kernel)


plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_4),cmap = 'gray')
plt.title(f'Convolved image by DoG kernel with  \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\theta$,size) =
↪  {std_l,std_w,phi,Lambda,60,size}')
plt.axis('off')
plt.show()
```

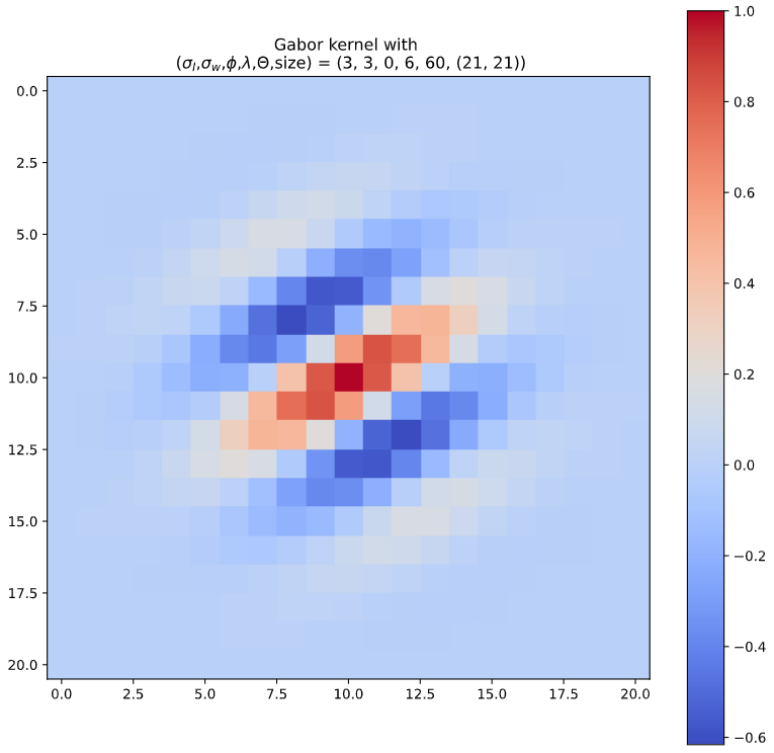

FIGURE 31. Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,60,(21,21))$

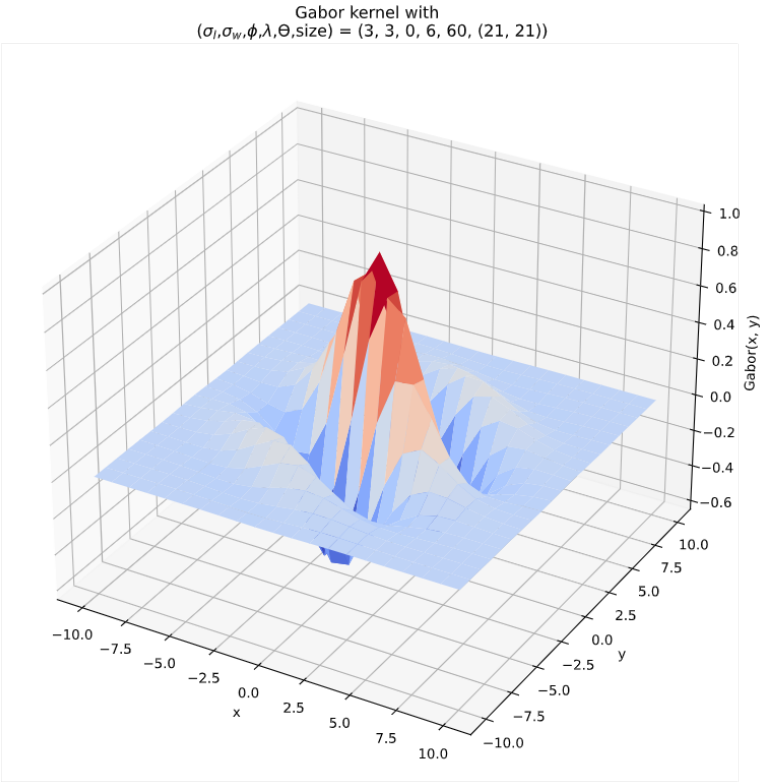

FIGURE 32. Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,60,(21,21))$

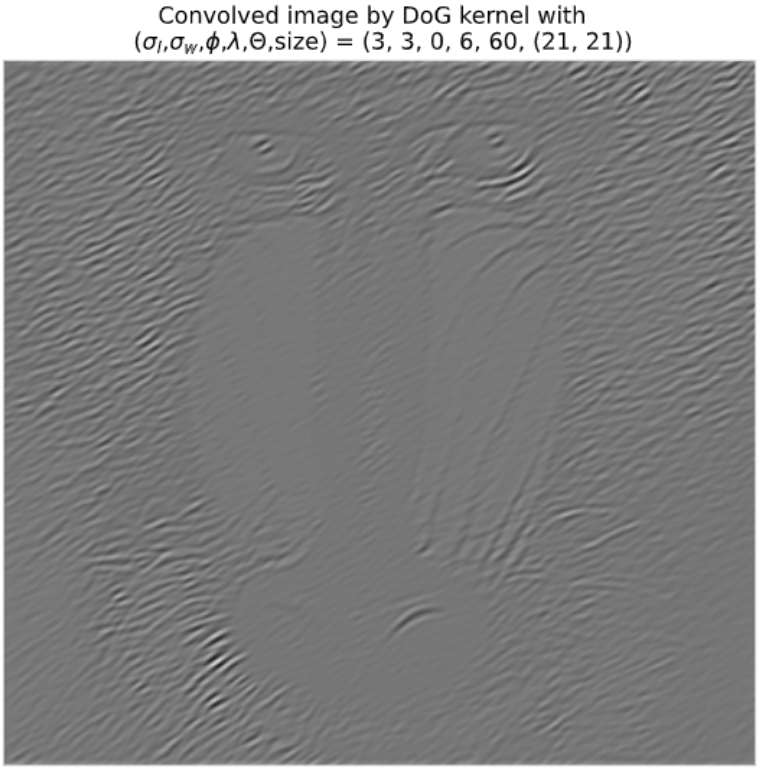

FIGURE 33. Convolved image with Gabor kernel with $(\sigma_l,\sigma_w,\phi,\lambda,\theta,\text{size}) = (3,3,0,6,60,(21,21))$

Let's combine the results we obtained in previous pages, to do that we need sum images after applying Gabor kernel with different orientations with $\theta = \pi/2,\pi/3,\pi/6,0$. Let's see the code for combining, plotting and edge detection operation.

```
composite_gabor = Gabor_result_1 + Gabor_result_2 + Gabor_result_3 +
↪  Gabor_result_4
img = np.array(min_max_scaler(composite_gabor) * 255, dtype = np.uint8)

plt.figure(figsize=(7,7))
plt.imshow(composite_gabor,cmap = 'gray')
plt.title('Output of combined Gabol kernels')
plt.axis('off')
plt.show()

plt.figure(figsize=(7,7))
plt.imshow(manuel_thresholding(img,127),cmap = 'gray')
plt.title('Thresholded image of output of combined Gabol kernels')
plt.axis('off')
plt.show()
```

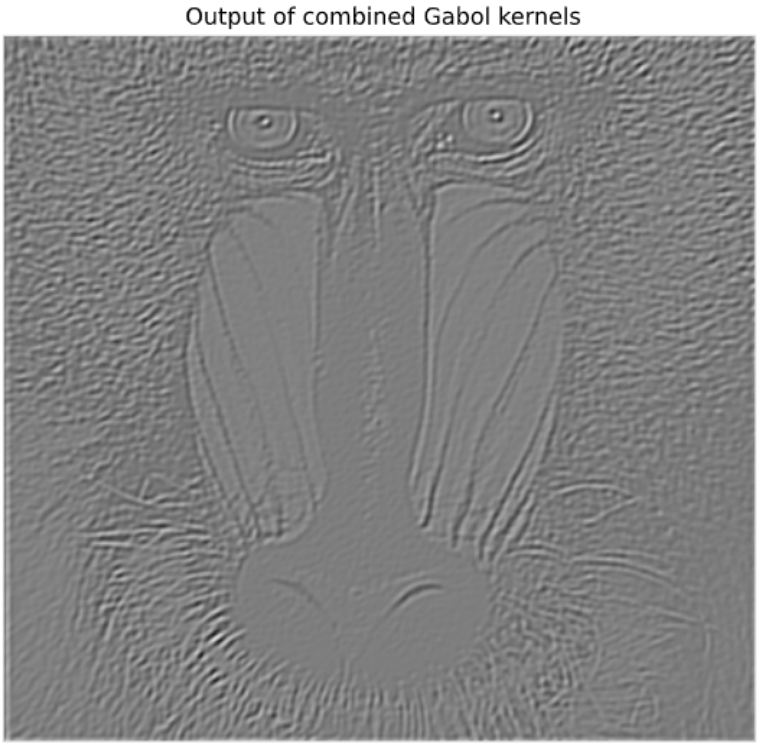

FIGURE 34. Output of combined results of Gabor responses

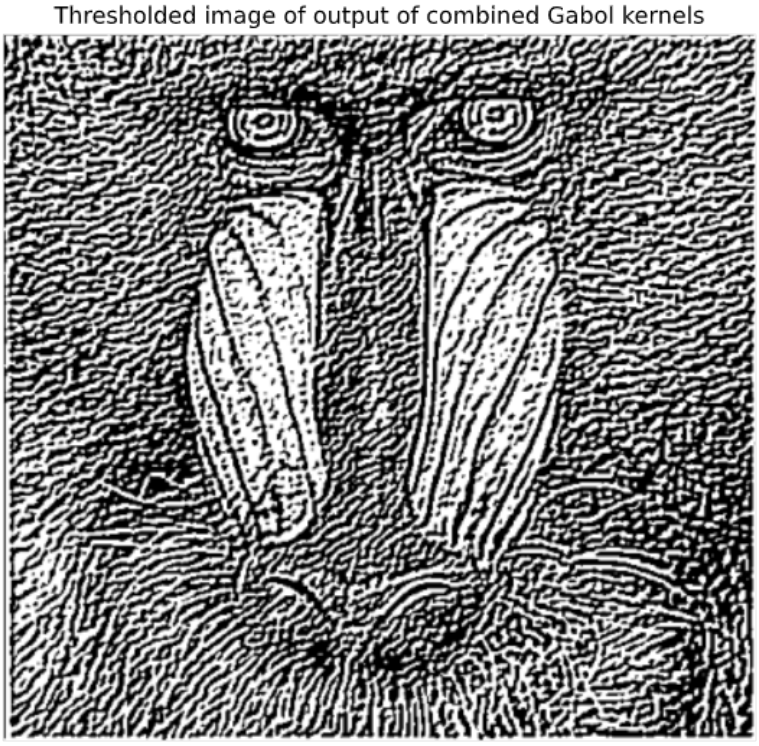

FIGURE 35. Thresholded image obtained by the output of combined results of Gabor responses

Let's also make a grid of figure that consist of image obtained by the output of combined results of Gabor responses and DoG kernel, just for sake of the simplicity for comparison.

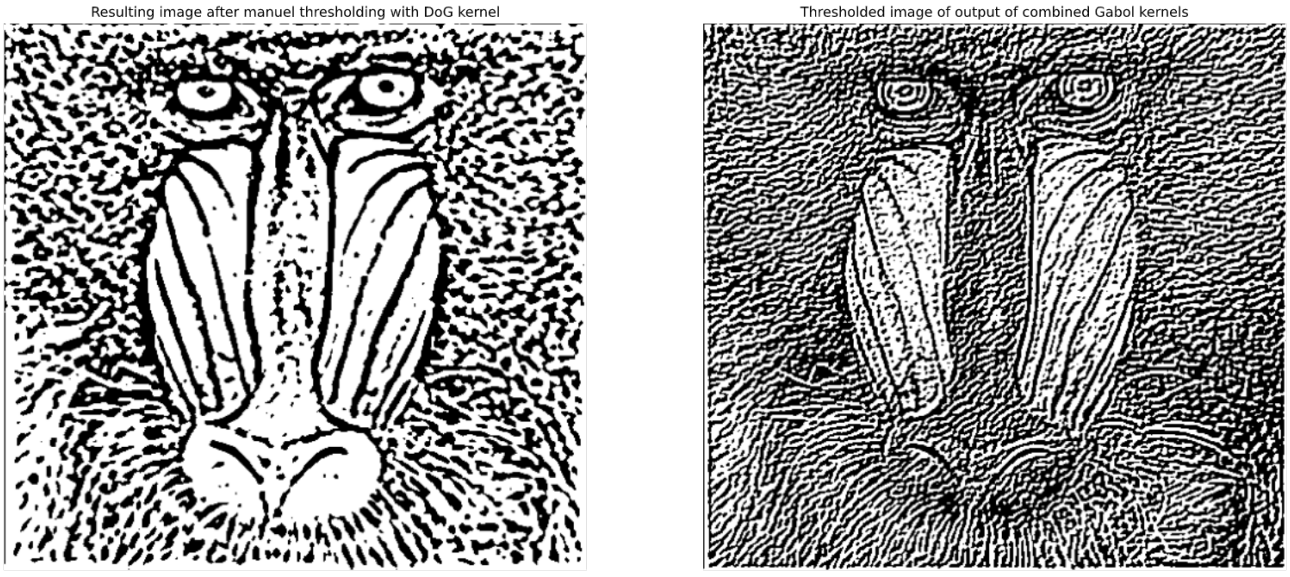

FIGURE 36. Comparison figure between image obtained by the output of combined results of Gabor responses and DoG kernel

As we can see clearly from the grid figure from above, the combination of Gabor filters responses performed much more better than single DoG kernel from the edge detection & feature extraction performance. There are various reasons for that such as Gabor filters considers the orientation and also we combine 4 different orientations. The extracted features, in our case, mostly edges, from the Gabor filters are spatially better than the ones extracted from the DoG kernel since we can see the neighbourhood relations much more clearly. As a further improvements onto Gabor filters, the rule of thumb approach in real image processing application is the usage of bank of Gabor filters as we did in this part. So, we can add much more orientations then construct a bank of Gabor filters to improve the features quality. Also, as we saw, there are parameters of Gabor filters such as π,σ,λ, etc., that can be further tuned to make improvements. Specifically, we can tune the λ to change the wavelength of the sinusoidal components, θ to get different orientations of the normal to the parallel stripes of Gabor kernel. Also, we have a phase effect ϕ so that we can give more/less offset to sinusoidal function to get better results. As a Gaussian part of the Gabor, we can also tune the σ parameters to arrange the envelope of the Gaussian. Furthermore, some Gabor filters uses additional φ parameter to adjust the spatial aspect ratio and ellipticity of the support of the Gabor kernel, i.e., new 2-D Gabor function becomes with the new parameter φ as follows

$$D(\vec{x}) = \exp\left(-(\vec{k}(\theta)\cdot\vec{x})^2/2\sigma_l^2 - (\vec{k_\perp}(\theta)\cdot\varphi\vec{x})^2/2\sigma_w^2\right)\cos\left(2\pi\frac{\vec{k_\perp}(\theta)\cdot\vec{x}}{\lambda}+\phi\right) \tag{6}$$

Briefly, let's sum it up in one paragraph. Increasing the wavelenth θ produces a thicker stripes that can be effective in some situations such that the extracted features quality increases. We can create different orientations of Gabor function, we can tune the φ to control over the aspect ratio so that it fits into specific situation then we can also tune the bandwith of the Gabor function σ to arrange the envelope of the Gaussian component of Gabor and much more. Finally, experimenting with the parameters of the Gabor kernel and its application with the bank of Gabor filters will eventually extract spatially meaningful informations from the natural images that can have random Gaussian noise in real life situations.

3. SOURCE CODE

```
# -*- coding: utf-8 -*-
"""EEE482_Can_Kocagil_21602218_Hw_2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/14TTuu1Vul3FC5lRg3JCOQHyy4D4JDL50
"""

import numpy as np
import scipy.io as io
import matplotlib.pyplot as plt

#pwd

#ls

data = io.loadmat('c2p3.mat')
print(data.keys())

counts = data['counts']
stim = data['stim']

print(f"The shape of the counts data {counts.shape}")
print(f"The shape of the Stimilus data {stim.shape}")

def STA(stim:np.ndarray,rho:np.ndarray,num_timesteps:int) -> np.ndarray:
    """
        Given the stimilus, rho (spike train) and number of time steps, computes
        Spike-Triggered-Average (STA). The spike-triggered average (STA) is a
    ↪  measure to relate a                    continuous signal and a
    ↪  simultaneously recorded spike train. It represents the average  signal
    ↪  taken         at the times of spike occurrences and with proper
    ↪  normalization is equivalent to the   cross-correlation between the
    ↪  continuous signal and the spike  -train. The STA provides an estimate of a
    ↪  neuron's linear receptive field.


            Arguments:
                - stim (np.ndarray)   : Stimulus data to be subjected
                - rho  (np.ndarray)   : Time-series Spike-train data
                - num_timesteps (int) : Number of timesteps before spike

            Returns:
                - _STA (np.ndarray)    : Averaged Stimulis data taken at spike
    ↪  occurrences



    """

```

```
    # Creating zero matrix for STA:
    stim_h,stim_w = stim.shape[:2]
    _STA = np.zeros((stim_h,stim_w,num_timesteps))

    # Finding spike times + num_timesteps
    spike_times = rho[num_timesteps:].nonzero()[0] + num_timesteps


    print(f'There are {len(spike_times)} of spikes.')

    for idx_spike in spike_times:
        _STA += stim[:,:,idx_spike - num_timesteps:idx_spike]

    _STA = _STA.astype(np.float)
    _STA /= len(spike_times)

    return _STA

num_timesteps = 10
sta = STA(stim,counts,num_timesteps)

kwargs = dict(
    cmap = 'gray',
    vmin = sta.min(),
    vmax = sta.max()
)


for i in np.arange(num_timesteps):
    plt.figure()
    plt.imshow(sta[:,:,i],**kwargs)
    plt.title(f"{num_timesteps - i} steps before the spike")
    plt.axis('off')
    plt.savefig(f"{num_timesteps - i}.png")

fig,axs = plt.subplots(1,2,figsize = (20,40))
axs[0].imshow(np.sum(sta,axis = 0),**kwargs)
axs[0].set_title(f"Row sum of STA")
axs[1].imshow(np.sum(sta,axis = 1),**kwargs)
axs[1].set_title(f"Column sum of STA")


plt.show()
plt.savefig('SpatialDimensionSummationofSTANew' + '.png')

project_stim = np.array([np.sum(sta[:,:, num_timesteps - 1] * _stim) for _stim
↪  in np.swapaxes(stim,0,2)])
project_stim /= project_stim.max()
kwargs = dict(
    bins=100,
    alpha=.8,
    rwidth=.66,

)
plt.figure(figsize=(10, 5))
```

```
plt.hist(project_stim,color='g', **kwargs)
plt.title('Histogram of Stimilus Projections on STA')
plt.ylabel('Spike Frequency')
plt.xlabel('Normalized Stimulus Projection')
plt.show()
plt.savefig('HistogramSTA.png')

spike_times = counts.nonzero()[0]

project_stim_nonzero = np.array([np.sum(sta[:,:, num_timesteps - 1] *
↪  stim[:,:,i]) for i in spike_times])
project_stim_nonzero /= project_stim_nonzero.max()


plt.figure(figsize=(10, 5))
plt.hist(project_stim_nonzero, **kwargs,color='orange')
plt.title('Histogram of Stimilus Projections on STA for non-zero spikes')
plt.ylabel('Spike Frequency')
plt.xlabel('Normalized Stimulus Projection')
plt.show()

plt.figure(figsize=(10, 5))
plt.hist(project_stim, **kwargs)
plt.title('Comparison Histogram of Stimilus Projections on STA')
plt.hist(project_stim_nonzero, **kwargs)
plt.ylabel('Spike Frequency')
plt.xlabel('Normalized Stimulus Projection')
plt.legend(['For all Spikes','For nonzero spikes'])
plt.show()

def DoG_Receptive_Field(std_c:int,std_s:int,size:int) -> np.ndarray:
    """
        Construct an on-center difference-of-gaussians (DoG) center-surround
↪  receptive field
        centered at 0 with the given size. Generally, the Difference of Gaussian
↪  module is a             filter that  identifies edges.Additionally, DoG is a
↪  feature enhancement algorithm that          involves the subtraction of one
↪  Gaussian blurred version of an original image from               another,
↪  less blurred version of the original


            Arguments:
                - std_c (int) : Standard deviation of lower sigma
                - std_s (int) : Standard deviation of higher sigma
                - size  (int) : Size of DoG kernel

            Returns:
               - DoG_kernel (np.ndarray) : DoG kernel with given (size,size)

    """


    import math

```

```
    # Computing lower sigma kernel:
    low_sigma_kernel  = np.fromfunction(
        lambda x, y: (1/(2*math.pi*std_c**2)) * math.e ** ((-1*((x-(size-1)/2)**2+(y-(size-1)/2)**2))/(2*std_c**2)), (size, size)
        )

    # Computing higher sigma kernel:
    high_sigma_kernel = np.fromfunction(
        lambda x, y: (1/(2*math.pi*std_s**2)) * math.e ** ((-1*((x-(size-1)/2)**2+(y-(size-1)/2)**2))/(2*std_s**2)), (size, size)
        )

    # DoG kernel:
    DoG_kernel = low_sigma_kernel - high_sigma_kernel


    return DoG_kernel

DoG_kernel = DoG_Receptive_Field(**
dict(
    size = 21,
    std_c = 2,
    std_s = 4

))

def plotFilter2D(kernel:np.ndarray,title:str,
                 cmap:str='coolwarm',
                 figsize:tuple = (10,10),
                 kwargs:dict = None) -> None:

    """
        Given the kernel, plot the kernel in 2-D surface.

            Arguments:
                - kernel (np.ndarray) : Kernel to be plotted
                - title  (str)        : Title of the figure
                - cmap   (str)        : Texture of the figure
                - figsize (tuple)     : Figure size
                - kwargs (dict)       : Additional orguments to plot if exists

            Returns:
                - None



    """

    plt.figure(figsize = figsize)

    if kwargs is not None:
        plt.imshow(kernel,cmap=cmap,**kwargs)
    else:
        plt.imshow(kernel,cmap=cmap)
```

```
199
200     plt.title(title)
201     plt.colorbar()
202     plt.show()
203
204
205
206
207 def plotFilter3D(kernel:np.ndarray,title:str,
208                  kernel_name:str,
209                  cmap:str='coolwarm',
210                  figsize:tuple = (10,10),
211                  kwargs:dict = None) -> None:
212
213     """
214         Given the kernel, plot the kernel in 2-D surface.
215
216             Arguments:
217                 - kernel (np.ndarray) : Kernel to be plotted
218                 - title  (str)        : Title of the figure
219                 - kernel_name (str)   : The name of the filter/kernel
220                 - cmap   (str)        : Texture of the figure
221                 - figsize (tuple)     : Figure size
222                 - kwargs (dict)       : Additional orguments to plot if exists
223
224             Returns:
225                 - None
226
227
228
229     """
230
231     fig_3D = plt.figure(figsize = (10,10))
232     size = kernel.shape[0]
233     ax_3D = plt.axes(projection='3d')
234     x,y = np.meshgrid(
235         np.arange( - int(size / 2), 1 + int(size / 2)),
236         np.arange( - int(size / 2), 1 + int(size / 2))
237                 )
238
239
240     if kwargs is not None:
241         ax_3D.plot_surface(x, y, kernel,cmap = cmap, **kwargs)
242     else:
243         ax_3D.plot_surface(x, y,kernel, cmap = cmap)
244
245     ax_3D.set_xlabel('x')
246     ax_3D.set_ylabel('y')
247     ax_3D.set_zlabel(f'{kernel_name}(x, y)')
248     plt.title(title)
249     plt.show()
250
251 kwargs = dict(
252     rstride=1,
253     cstride=1,
```

```
    edgecolor='none'
)


plotFilter2D(DoG_kernel,f'Difference of Gaussian (DoG) kernel with Stds {(2,4)} in 21x21 form')

plotFilter3D(DoG_kernel,f'Difference of Gaussian (DoG) kernel with Stds {(2,4)} in 21x21 form','DoG', kwargs = kwargs)

image = plt.imread('hw2_image.bmp')
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.axis('off')
plt.show()

class _Window(object):
  """
    This class creates a generator that slide over the given image with
    predefined step and window size.

      Attributes:
        - image      : input image to be sliding
        - step_size  : it determines how much slice the generator extract
        - dims       : resulting image's dimensions

      Methods:
        - __iter__   : creates a generator over the image with given patch dimensions

  """
  def __init__(self,image : np.ndarray, step_size:tuple, dims: tuple):
    """
      Creates an constructor for _Window class, this method encapsulates
      all necessary data to generate windows over image.

    """
    self.image = image
    self.dims = dims
    self.step_size = step_size

  def __iter__(self):
    """ Generator function to window over image. """
    for x in range(self.dims[0]):
      for y in range(self.dims[1]):
        yield self.image[x:x + self.step_size[0], y:y + self.step_size[1]]


def Conv2D(source_image : np.ndarray, kernel: np.ndarray) -> np.ndarray:
  """
   Convolution is the process of adding each element of the image to its local neighbors,           weighted by the kernel. This is related to a form of mathematical convolution.

```

```
303       Arguments:
304         - source_image   (np.ndarray) : Gray scale image to be convolved
305         - kernel         (np.ndarray) : Kernel to be sliding over the image
306 
307       Returns:
308         - conv_image     (np.ndarray) : Resulting convolved image
309   """
310 
311   assert (len(source_image.shape)   == 2), "Image is not gray scale"
312   assert (len(kernel.shape) == 2), "Kernel is not in required size"
313 
314   source_image =  np.asarray(source_image).clip(0,255).copy()
315   H,W = source_image.shape
316   k_h,k_w = kernel.shape
317   padded_image = np.pad(array = source_image, pad_width = max(k_w,k_w) // 2 + 1,
      ↪  mode = 'constant')
318 
319   new_H,new_W = padded_image.shape
320   h_pad , w_pad  =  (new_H - H) , (new_W - W)
321 
322   # Creating sliding window:
323   image_window = _Window(padded_image,(k_h,k_w),(new_H - h_pad, new_W - w_pad))
324 
325   #kernel = np.flipud(np.fliplr(kernel))
326   # Main operation:
327   conv_image = [(patch * kernel).sum() for patch in image_window]
328 
329   return np.array(conv_image).reshape(H,W)
330 
331 
332 
333 def min_max_scaler(matrix : np.ndarray)-> np.ndarray :
334   """
335     Given the matrix, apply normalization as follow:
336 
337         norm_matrix = (matrix - min_val) / (max_val - min_val)
338 
339     Arguments:
340       -  matrix     (np.ndarray) : Input Matrix
341 
342     Returns:
343       - norm_matrix (np.ndarray) : Normalized version of matr
344 
345   """
346   matrix = np.asarray(matrix).copy()
347   max_val = matrix.max()
348   min_val = matrix.min()
349 
350   return (matrix - min_val) / (max_val - min_val)
351 
352 if (image[:,:,0] == image[:,:,1]).all() and \
353    (image[:,:,0] == image[:,:,2]).all() and \
354    (image[:,:,1] == image[:,:,2]).all():
355 
356    print('True')
```

```

gray_image = image[:,:,0]
filtered_image = Conv2D(gray_image,DoG_kernel)

plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(filtered_image),cmap = 'gray')
plt.title('Convolved image by DoG kernel')
plt.axis('off')
plt.show()

def otsu_threshold(source_image: np.ndarray) -> np.ndarray:
    """
    Otsu's automatic thresholding method that seperates background and
    foreground of image with automatically computed threshold value. This
    threshold is computed via minimizing the within-class varicance or
    maximing the inter-class variability that yields same results. However,
    note that computed threshold value by maximizing the inter-class variance
    is more computationally efficient method.

      Arguments:
        - source_image (np.ndarray) : Source image to be thresholded by Otsu

      Returns:
        - thres_image  (np.ndarray) : Resulting image after applying Otsu's
↪  method


  """

    assert (len(source_image.shape) == 2), "Image is not gray scale"

    source_image =  np.asarray(source_image).clip(0,255).copy()

    hist =  [np.sum(pixel == source_image) for pixel in np.arange(256)]


    otsu_thres = 0
    var = 0
    max_val = 256

    for thres in np.arange(256):

      # For first class, mean and variance:
      w_0 = sum(pixel for pixel in hist[:thres])
      mu_0 = sum([i * hist[i] for i in range(thres)]) / w_0 if w_0 > 0 else 0

      # For second class, mean and variance:
      w_1 = sum(pixel for pixel in hist[thres:])
      mu_1 = sum([j * hist[j] for j in range(thres, max_val)]) / w_1 if w_1 > 0
      ↪  else 0

      # calculating inter-class variance
      s = w_0 * w_1 * (mu_0 - mu_1) ** 2
```

```
409
410       # Set threshold if new inter class variance is bigger than previous
411       otsu_thres = thres - 1 if s > var else otsu_thres
412       var = s if s > var else var
413
414     print(f'Self written Otsu threshold value is {otsu_thres}')
415
416     # Apply thresholding: Hence
417     source_image[source_image >= otsu_thres] = 255
418     source_image[source_image < otsu_thres] = 0
419
420     return source_image.astype(np.uint8)
421
422 def manuel_thresholding(source_image: np.ndarray,threshold:int) -> np.ndarray:
423     """
424         Given the source image and threshold, apply thresholding (i.e., setting
    ↪  all values
425         above a certain threshold to 1 and the remainder to 0.
426
427             Arguments:
428                 - source_image (np.ndarray) : Source image to be thresholded
429                 - threshold    (int)        : Threshold value
430
431             Returns:
432                 - thres_image  (np.ndarray) : Resulting image after applying
    ↪  threshold
433
434
435
436     """
437
438
439     assert (len(source_image.shape) == 2), "Image is not gray scale"
440
441     source_image =  np.asarray(source_image).clip(0,255).copy()
442
443
444
445     # Apply thresholding:
446     source_image[source_image >= threshold] = 255
447     source_image[source_image < threshold] = 0
448
449     return source_image.astype(np.uint8)
450
451 img = np.array(min_max_scaler(filtered_image) * 255, dtype = np.uint8)
452
453
454 plt.figure(figsize=(7,7))
455 plt.imshow(manuel_thresholding(img,115),cmap = 'gray')
456 plt.title('Resulting image after manuel thresholding')
457 plt.axis('off')
458 plt.show()
459
460
```

```
img = np.array(min_max_scaler(filtered_image) * 255, dtype = np.uint8)
plt.figure(figsize=(7,7))
import cv2
plt.imshow(cv2.threshold(img,0,255,cv2.THRESH_BINARY+cv2.THRESH_OTSU)[1],cmap = 'gray')
plt.title('Resulting image after otsu thresholding')
plt.axis('off')
plt.show()



img = np.array(min_max_scaler(filtered_image) * 255, dtype = np.uint8)
plt.figure(figsize=(7,7))
plt.imshow(otsu_threshold(img),cmap = 'gray')
plt.title('Convolved image by DoG kernel')
plt.axis('off')
plt.show()

def Gabor_Receptive_Field(std_l:int,std_w:int,
                          theta:float,Lambda:int,
                          psi:int, size:tuple) -> np.ndarray:
    """
        Gabor Receptive Field is a linear filter used for texture analysis, which essentially means that it analyzes whether there is any specific frequency content in the image in specific directions in a localized region around the point or region of analysis


            Arguments:
                - std_l  (int)   : Standard deviation of lower sigma
                - std_w  (int)   : Standard deviation of higher sigma
                - theta  (float) : Orientation
                - Lambda (int)   : Gabor filter parameter
                - psi    (int)   : Gabor filter parameter
                - size   (tuple) : Size of Gabor kernel

            Returns:
               - Gabor_kernel (np.ndarray) : Gabor kernel with given (size,size)

    """


    import math

    x, y = np.meshgrid(np.arange(- int(size[0] / 2), 1 + int(size[0] / 2)),
                       np.arange(- int(size[1] / 2), 1 + int(size[1] / 2)))


    # Rotation
    x_theta = x * np.cos(theta) + y * np.sin(theta)
    y_theta = -x * np.sin(theta) + y * np.cos(theta)

    # Main function for creating Gabor filter:
```

```
    Gabor_kernel = np.exp(-.5 * (x_theta ** 2 / std_l ** 2 + y_theta ** 2 / std_w ** 2)) * np.cos(2 * np.pi / Lambda * x_theta + psi)


    return Gabor_kernel

theta_90 = np.pi / 2
size = (21,21)
kwargs_kernel = dict(
        std_l = 3,
        std_w = 3,
        psi = 0,
        Lambda = 6,
        size = size
)

std_l = 3
std_w = 3
phi = 0
Lambda = 6
size = size

Gabor_90_kernel = Gabor_Receptive_Field(theta = theta_90,**kwargs_kernel)


kwargs = dict(
    rstride=1,
    cstride=1,
    edgecolor='none'

)

plotFilter2D(Gabor_90_kernel,f'Gabor kernel with \n ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) = {std_l,std_w,phi,Lambda,90,size}'
)



plotFilter3D(Gabor_90_kernel,f'Gabor kernel with \n ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) = {std_l,std_w,phi,Lambda,90,size}',kernel_name = 'Gabor',kwargs=kwargs)


Gabor_result_1 = Conv2D(image[:,:,0],Gabor_90_kernel)

plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_1),cmap = 'gray')
plt.title('Convolved image by DoG kernel')
plt.axis('off')
plt.show()

theta_0 = 0
Gabor_0_kernel = Gabor_Receptive_Field(theta  = theta_0, **kwargs_kernel)

```

```
plotFilter2D(Gabor_0_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,theta_0,size}')

plotFilter3D(Gabor_0_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,theta_0,size}',kernel_name = 'Gabor',kwargs=kwargs)


Gabor_result_2 = Conv2D(image[:,:,0],Gabor_0_kernel)


plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_2),cmap = 'gray')
plt.title(f'Convolved image by DoG kernel with  \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,0,size}')
plt.axis('off')
plt.show()

theta_30 = np.pi / 6

Gabor_30_kernel = Gabor_Receptive_Field(theta = theta_30,**kwargs_kernel)
plotFilter2D(Gabor_30_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,30,size}')

plotFilter3D(Gabor_30_kernel,f'Gabor kernel with \n
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,30,size}',kernel_name = 'Gabor',kwargs=kwargs)


Gabor_result_3 = Conv2D(image[:,:,0],Gabor_30_kernel)


plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_3),cmap = 'gray')
plt.title(f'Convolved image by DoG kernel with
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,30,size}')
plt.axis('off')
plt.show()

theta_60 = np.pi / 3

Gabor_60_kernel = Gabor_Receptive_Field(theta = theta_60,**kwargs_kernel)

plotFilter2D(Gabor_60_kernel,
title = f'Gabor kernel with
↪  ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) =
↪  {std_l,std_w,phi,Lambda,60,size}'

)

```

```
plotFilter3D(Gabor_60_kernel,f'Gabor kernel with ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) = {std_l,std_w,phi,Lambda,60,size}',kernel_name = 'Gabor',kwargs=kwargs)



Gabor_result_4 = Conv2D(image[:,:,0],Gabor_60_kernel)


plt.figure(figsize=(7,7))
plt.imshow(min_max_scaler(Gabor_result_4),cmap = 'gray')
plt.title(f'Convolved image by DoG kernel with ($\sigma_l$,$\sigma_w$,$\phi$,$\lambda$,$\Theta$,size) = {std_l,std_w,phi,Lambda,60,size}')
plt.axis('off')
plt.show()

composite_gabor = Gabor_result_1 + Gabor_result_2 + Gabor_result_3 + Gabor_result_4
img = np.array(min_max_scaler(composite_gabor) * 255, dtype = np.uint8)



plt.figure(figsize=(7,7))
plt.imshow(composite_gabor,cmap = 'gray')
plt.title(' Output of combined Gabol kernels')
plt.axis('off')
plt.show()

plt.figure(figsize=(7,7))
plt.imshow(manuel_thresholding(img,127),cmap = 'gray')
plt.title('Thresholded image of output of combined Gabol kernels')
plt.axis('off')
plt.show()
```

REFERENCES

[1] Junji Ito. "Spike Triggered Average". In: *Encyclopedia of Computational Neuroscience*. Ed. by Dieter Jaeger and Ranu Jung. New York, NY: Springer New York, 2015, pp. 2832–2835. ISBN: 978-1-4614-6675-8. DOI: 10.1007/978-1-4614-6675-8_407. URL: https://doi.org/10.1007/978-1-4614-6675-8_407.

[2] Wikipedia contributors. *Difference of Gaussians — Wikipedia, The Free Encyclopedia*. [Online; accessed 1-March-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Difference_of_Gaussians&oldid=1007271206.

[3] Wikipedia contributors. *Gabor filter — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Gabor_filter&oldid=993157632. [Online; accessed 2-March-2021]. 2020.

[4] Wikipedia contributors. *Spike-triggered average — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Spike-triggered_average&oldid=1000479639. [Online; accessed 1-March-2021]. 2021.