Tarea 1:
Detección de objetos en imágenes basado en modelo de Hubel y Wiesel

Fecha de entrega: Martes 12 de octubre, 23:59 hrs.

Estudiante:	Francisco Molina L.
Profesor:	Claudio A. P.
Auxiliar:	Jorge Zambrano I.
Ayudantes:	Eitan Hasson A.
	Juan Pérez C.
Semestre:	Primavera 2021

Índice

1. Introducción

En el presente informe se desarrollan las actividades correspondientes a la Tarea 1 del curso EL7007-1 del semestre de primavera de 2021. Este informe comprende la implementación computacional de algoritmos de detección de patrones específicos y generales mediante el esquema de Hubel y Wiesel.

Los principales objetivos de este informe corresponden a: lograr implementar mediante programación la implementación de la convolución, del proceso de binarizar una imagen, de recortar patrones interactivamente, de calcular filtros detectores y de elegir el umbral de detección interactivamente. Pero principalmente, se desea lograr detectar los patrones solicitados en las imágenes entregadas por el equipo docente.

En las **Secciones 2 y 3** se detallan las funciones implementadas en Python, adjuntas en el notebook Tarea1.ipynb, necesarias para la detección de patrones. En la **Sección 4** se presentan los resultados obtenidos de la implementación en Python, en la **Sección 5** se realiza un análisis de dichos resultados, destacando los casos interesantes, y en la **Sección 6** se presentan las conclusiones del informe.

2. Detección de un patrón específico

En esta sección se desarrolla un sistema de detección de un patrón simple. En particular, la detección de una L dibujada en un *array* de (10x10), formada por 3 pixeles verticales y dos horizontales, como se muestra en la **Figura** 4.

2.1. Calculo de la convolución

Para esto se necesita implementar una función que realice la convolución entre una imagen y un kernel, ambos de tamaño arbitrario. Calcular una convolución consiste en usar una ventana deslizante del mismo tamaño del kernel y calcular el producto punto entre los pixeles debajo de la ventana y el kernel, como se muestra en la **Figura** 1:

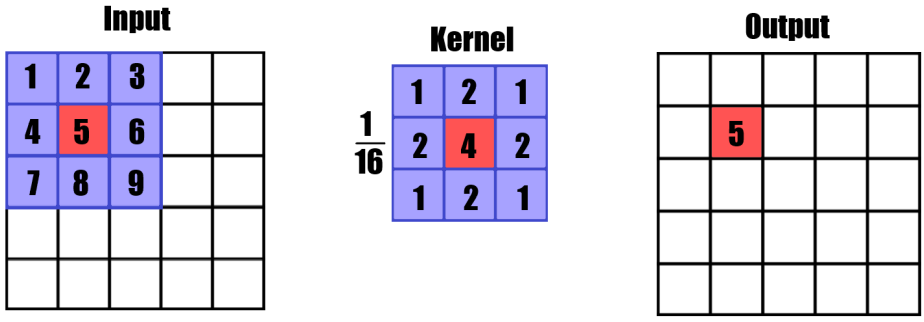

Figura 1: Ejemplo de convolución

Realizar la convolución de este modo implica que la imagen resultante será de menor tamaño, pues el kernel se sale de la imagen en los bordes. En general, para una imagen de dimensiones (N,N) y un kernel de dimensiones (k,k), la imagen resultante tiene un tamaño de:

$$(N - k + 1, N - k + 1)$$

Esto puede evitarse aplicando **padding**, rellenar los pixeles que quedan fuera de la imagen con algún valor arbitrario, como se muestra en la **Figura** 2:

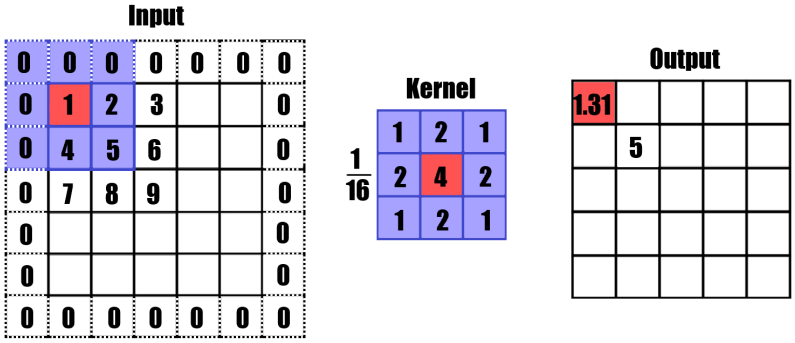

Figura 2: Convolución con zero padding

De esta forma, el kernel se puede aplicar en todos los pixeles de la imagen y se logra preservar el tamaño original de esta.

Para implementar la convolución en Python es necesario utilizar 4 *for loops*. Dos para recorrer, pixel a pixel, la imagen original, y los otros dos para recorrer el kernel cuando ya se está en el pixel central. Sin embargo, esto no contempla cómo se recorre la sub-sección de la imagen bajo el kernel (la sombra del kernel), es decir, cómo se encuentran los pixeles que hay que multiplicar con el kernel.

Para esto en la implementación se utilizan ***offsets***, valores que indican cuantos pixeles hay que desplazarse desde el pixel central, tanto en X como en Y, para llegar a la esquina superior izquierda de la sombra del kernel.

Los kernels son de dimensiones impares, para que así puedan tener un pixel central. Analizando el caso del kernel 3x3 y el kernel 5x5, como se muestra en la **Figura 3**:

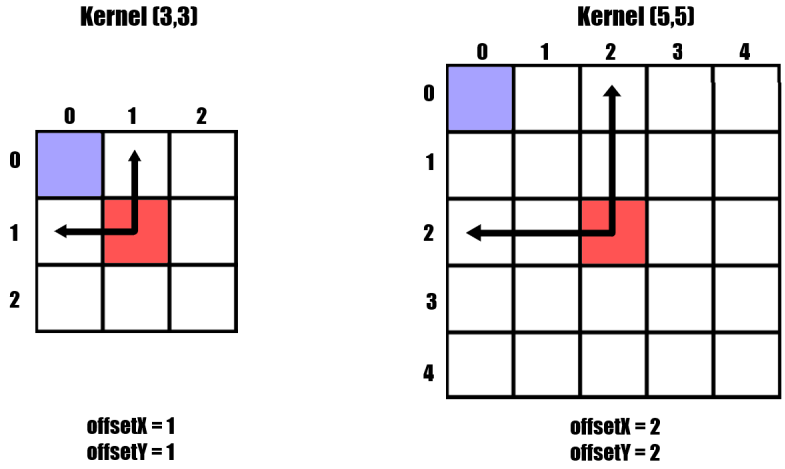

Figura 3: Offsets

Es fácil darse cuenta que el pixel en la esquina superior izquierda del kernel de dimensiones (k,k) siempre está a una distancia:

$$(\left\lfloor \frac{k}{2} \right\rfloor, \left\lfloor \frac{k}{2} \right\rfloor)$$

del kernel central.

Siguiendo estas ideas la función de convolución realizada es la que se muestra en el **Listing 1**:

```
def convolution(image, kernel):
    output = np.zeros([image.shape[0],image.shape[1]])

    # number of rows and columns of input image
    rows = image.shape[0]
    cols = image.shape[1]

    # number of rows and columns of kernel
    kRows = kernel.shape[0]
    kCols = kernel.shape[1]

    # offset between central pixel of kernel and top left (beginning of for loop)
    offsetX = math.floor(kRows/2)
    offsetY = math.floor(kCols/2)

    # convolution
    for y in range(rows):
        for x in range(cols):
            sum = 0
            for kY in range(kRows):
                for kX in range(kCols):
                    posX = (x - offsetX) + kX
                    posY = (y - offsetY) + kY
                    # zero padding
                    if ((min(posX,posY) < 0) | (posX >= cols) | (posY >= rows)):
                        pixel = 0.0
                    else:
                        pixel = image[posY,posX]
                    sum += pixel*kernel[kY,kX]
            output[y,x] = sum
    return output
```

Listing 1: Función de convolución

2.2. Generar letra L, kernel balanceado y filtro detector

Lo siguiente es generar una matriz de (10x10) que contenga una letra L formada por tres pixeles verticales y dos horizontales, así como también un kernel balanceado de (3x3), es decir, un pixel con valor 8 al centro y -1 en el resto.

Para generar la matriz L se creó la función *makeL()*, la cual recibe el parámetro booleano *rand*, que si es *True* coloca la L en una posición aleatoria dentro de la matriz, y si no, la L se crea siempre partiendo en el pixel (4,4). Para crear la letra basta con crear una matriz de ceros de tamaño (10x10), seleccionar los pixeles deseados y reemplazarles su valor con 1, como se muestra en el **Listing 2**:

```
def makeL(rand):
    L = np.zeros((10,10))
    if rand:
        (y,x) = (random.randint(0,7),random.randint(0,8))
    else:
        (y,x) = (4,4)
    L[y:y+2,x] = 1
    L[y+2,x:x+2] = 1
    return L
```

Listing 2: Función generadora de la letra L

Para generar el kernel balanceado se creó la función *makeBalancedKernel*, que simplemente crea una matriz de ceros de tamaño (3x3) y rellena manualmente los valores correspondientes, como se muestra en el **Listing 3**:

```
def makeBalancedKernel():
    k = np.zeros((3,3))
    k[0:k.shape[0],0:k.shape[1]] = -1
    k[1,1] = 8
    return k
```

Listing 3: Función generadora del kernel balanceado

Al visualizar las matrices creadas con estas funciones mediante mapas de calor, como se muestra en la **Figura 4**, se puede ver que cumplen con los requerimientos especificados:

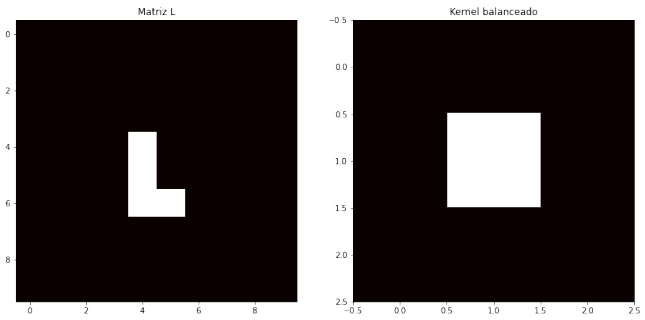

Figura 4: Matriz L y kernel balanceado

Para generar el filtro detector según el esquema de Hubel y Wiesel basta con realizar la convolución entre el kernel balanceado y el patrón que se esté buscando (en este caso la matriz L), y luego reflejar el resultado tanto horizontalmente como verticalmente. La función que se encarga de esto es *makeDetectionFilter()*, que recibe como entradas la imagen con el patrón *image* y el array del kernel balanceado *kernel*, como se muestra en el **Listing 4**:

```
def makeDetectionFilter(image, kernel):
    out = convolution(image, kernel)
    out = np.flip(out, 0)
    out = np.flip(out, 1)
    return out
```

Listing 4: Función generadora del filtro

El filtro resultante se puede visualizar usando mapas de calor, como se muestra en la **Figura 5**:

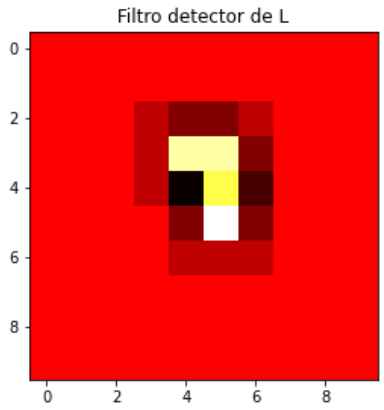

Figura 5: Filtro detector de L

2.3. Detección de la letra L

Para aplicar el filtro y detectar la letra L basta con calcular la convolución entre el filtro detector y la matriz con la L original, lo que se traduce al llamado de la función *convolution* mostrado en el **Listing 5**:

```
dFilter = makeDetectionFilter(L,k)
randomL = makeL(True)
detection = convolution(randomL, dFilter)
```

Listing 5: Detección letra L

Nótese que en este llamado el parámetro *rand* de la función *makeL* se escoge como *True*. Esto se hace para probar la consistencia del filtro, probando la detección de la letra L en distintas posiciones.

2.4. Presentar resultados mediante mapa de calor

El resultado de esta convolución se muestra en la **Figura 6**:

Figura 6: Detección de la letra L

Se puede apreciar que el pixel de mayor intensidad en la imagen de la detección se encuentra en la posición (5,6), que es precisamente donde se encuentra la esquina de la L en la imagen original

3. Detección de un patrón general

En esta sección se desarrolla un sistema de detección de patrones generales. En particular, se crea un algoritmo que permite seleccionar un patrón dentro de una imagen y que luego busca todas las instancias de dicho patrón mediante el esquema de Hubel y Wiesel.

3.1. Binarizar imagen

Para binarizar una imagen se debe recorrer esta, pixel por pixel, revisando si sus valores son mayores a algún umbral, en caso de que sí lo sean se anota un 1 en la imagen de salida en la posición correspondiente, y en caso contrario un cero. La función encargada de esta tarea es *makeBinary()* que recibe dos entradas, la imagen que se quiere binarizar *image* y el umbral *threshold*, se crea una matriz de ceros del mismo tamaño de la imagen de entrada y se sigue el procedimiento recién descrito, como se muestra en el **Listing 6**

```
1  def makeBinary(image, threshold):
2      image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
3      out = np.zeros(image.shape)
4      for y in range(image.shape[0]):
5          for x in range(image.shape[1]):
6              if image[y,x] >= threshold:
7                  out[y,x] = 1
8              else:
9                  out[y,x] = 0
10     return out
```

Listing 6: Función que binariza imágenes

Cabe destacar que es necesario trabajar con imágenes en escala de grises, de lo contrario sería ambiguo qué valor revisar para ver si es mayor que el umbral, por lo que en la primera linea de la función se añade un llamado a *cv2.cvtColor* para tarnsformar la imagen de RGB a escala de grises. Tomando como ejemplo la imagen de la carta del seis de diamantes, su binarización se muestra en la **Figura 7**:

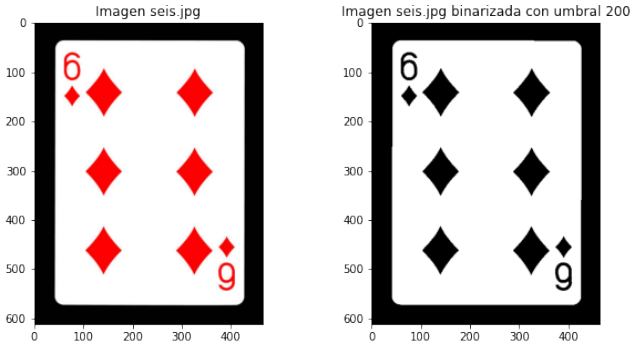

Figura 7: Binarización carta de seis de diamantes

El umbral se escoge de manera arbitraria, pero puede ser útil revisar el histograma de la imagen en escala de grises para decidirlo.

3.2. Recortar patrón de una imagen

Se crea un algoritmo que permite recortar un patrón de una imagen, señalando con el mouse la esquina superior izquierda y la esquina inferior derecha del rectángulo que contiene al patrón. Para esto se trabaja con la función *ginput* de *matplotlib*, que guarda información sobre los clics realizados en una ventana interactiva.

Para poder hacer uso de esta función en el entorno de *Jupyter Notebook*, es necesario cambiar el backend en el que trabaja *matplotlib*, en particular sirve "TkAgg". Al final del llamado de la función se debe restaurar el backend original mediante *%matplotlib inline* para que no haya problemas en las otras celdas.

La función encargada de esta tarea es *extractPattern*, que recibe como entrada la imagen binaria *image* de la cual se quiere extraer el patrón. Primero se muestra la imagen y luego se esperan 2 clics con *ginput*. Luego de esto se devuelve un recorte de la imagen original, en las posiciones indicadas por los clics, como lo muestra el **Listing 7**:

```
def extractPattern(image):
    # interactive window backend
    matplotlib.use('TkAgg')

    fig = plt.figure(figsize = (14,7))
    plt.imshow(image, cmap = 'gray')
    [P,Q] = np.array(plt.ginput(n = 2, timeout = 30), int)

    # original backend
    %matplotlib inline
    pattern = image[P[1]:Q[1], P[0]:Q[0]]
    plt.imshow(pattern, cmap = 'gray')
    return pattern
```

Listing 7: Función que extrae patrones

La ventana que despliega *ginput* se muestra en la **Figura 8**:

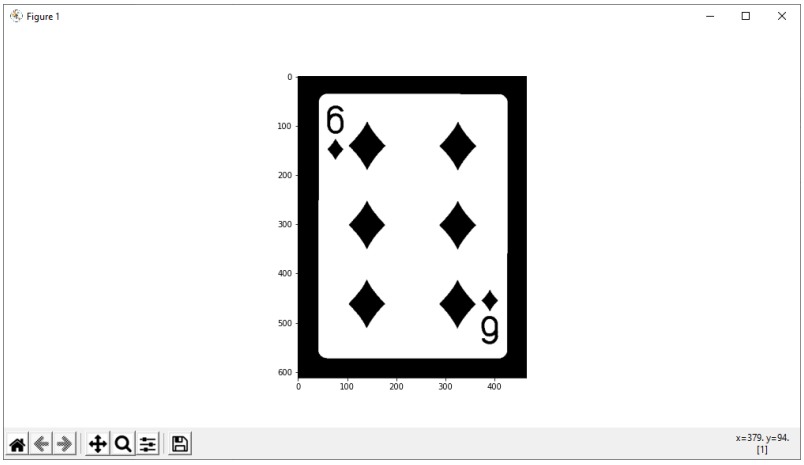

Figura 8: Ventana ginput para el seis de diamantes

Y el patrón extraído en el ejemplo anterior (un diamante) se muestra en la **Figura 9**:

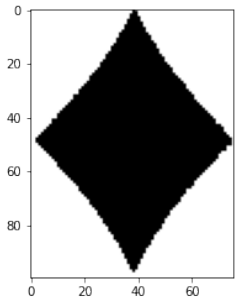

Figura 9: Patrón extraído para el seis de diamantes

3.3. Filtro detector

Al igual que en la sección 2.2, para generar el filtro detector basta con convolucionar el kernel balanceado con el patrón recortado y reflejar el resultado en el eje horizontal y el vertical. La función que se encarga de esta tarea es *getPatternFilter*, que recibe como entrada el array que contiene al patrón (*pattern*), la cual se muestra en el **Listing 8**:

```
def getPatternFilter(pattern):
    conv = convolution(pattern, makeBalancedKernel())
    conv = np.flip(conv, 1)
    conv = np.flip(conv, 0)
    plt.imshow(conv, cmap = 'hot')
    plt.title("Filtro que detecta el patron escogido")
    plt.show()
    return conv
```

Listing 8: Función que calcula el filtro detector

Siguiendo con el ejemplo del seis de diamantes, el filtro detector de diamantes obtenido utilizando este método se muestra en la **Figura 10**:

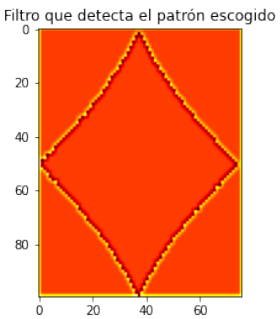

Figura 10: Filtro detector de diamantes

Con este filtro detector ya se puede realizar la detección en la imagen original, simplemente convolucionando el filtro con la imagen.

3.4. Umbral de detección

Al realizar la convolución entre un filtro detector y una imagen se va a obtener una matriz que va a tener valores máximos en las zonas donde se encuentra el patrón que se busca. Sin embargo, hay que escoger manualmente un umbral que permita decidir qué máximos son efectivamente los que sirven.

Este umbral se escoge analizando visualmente los valores de los pixeles del resultado de la convolución. Esto se hace "desenrollando" la matriz en un vector mediante *ravel()*, y haciendo un clic en el punto de corte deseado mediante *ginput*. Este método se implementa en la función *chooseTr()* que recibe como entrada el resultado de la convolución entre una imagen y un filtro detector *conv*. Esta se muestra en el **Listing 9**:

```
def chooseTr(conv):
    im = conv.ravel()
    # interactive window backend
    matplotlib.use('TkAgg')

    fig = plt.figure(figsize = (14,7))
    plt.plot(im)
    tr = np.array(plt.ginput(n = 1, timeout = 30), int)[0][1]

    # original backend
    %matplotlib inline

    plt.plot(im)
    plt.title("Valores detecciones con umbral")
    plt.hlines(tr, 0, len(conv.ravel()), 'red')
    return tr
```

Listing 9: Función para escoger el umbral

La ventana que despliega ginput se muestra en la **Figura 11**:

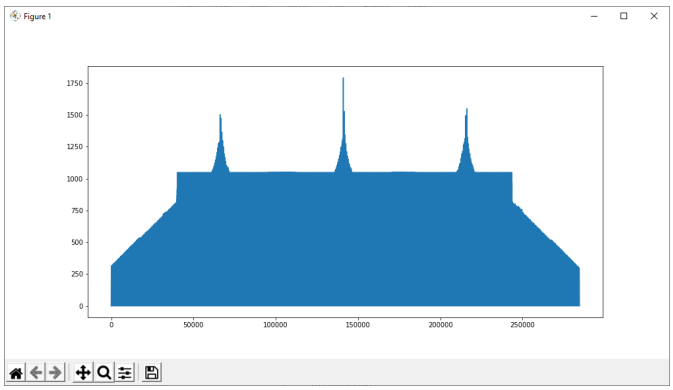

Figura 11: Ventana en la que se escoge el umbral de detección

Y al hacer clic a la altura a la que se quiere el umbral se guarda el valor del clic y se dibuja el umbral en la figura, como se muestra en la **Figura 12**:

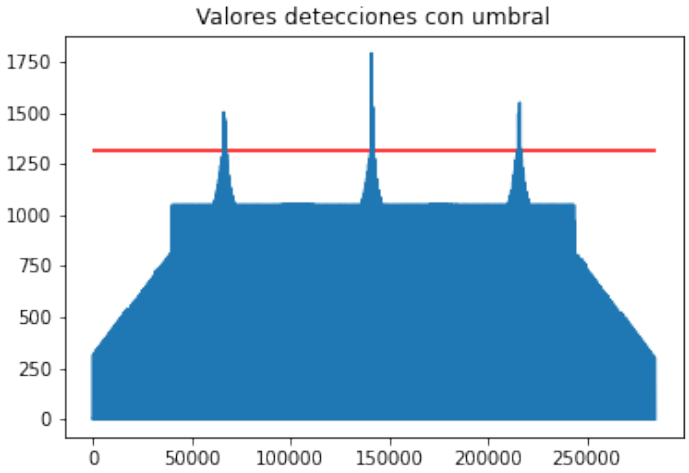

Figura 12: Umbral escogido

3.5. Recuperar posiciones de los máximos

Una vez escogido el umbral de detección, se deben recuperar las coordenadas (x,y) de los valores que superan el umbral. Para esto basta con recorrer la matriz, pixel a pixel, y guardar las coordenadas de los valores que superen el umbral.

La función que se encarga de esto es *getMaxima*, que recibe como entradas el resultado de la convolución entre una imagen y un filtro detector (*detection*) y el umbral escogido (*thresh*), la cual realiza el procedimiento recién descrito. Esta se muestra en el **Listing 10**:

```
def getMaxima(detection,thresh):
    maxima = []
    for y in range(detection.shape[0]):
        for x in range(detection.shape[1]):
            if detection[y,x] >= thresh:
                maxima.append({'coords':(x,y), 'value':detection[y,x]})
    return maxima
```

Listing 10: Función para recuperar máximos

Cabe destacar que se guarda en forma de diccionarios tanto las coordenadas del máximo como sus respectivos valores. Esto pues, los valores serán útiles para la función de supresión de no máximos.

3.6. Supresión de no máximos

Debido a que las detecciones se realizan mediante una convolución, y entre las posiciones en la que se aplica el kernel hay traslape, se van a encontrar múltiples detecciones para un mismo objeto, por lo que se necesita un algoritmo que logre filtrar múltiples detecciones de un mismo patrón en una sola, como se muestra en la **Figura 13**:

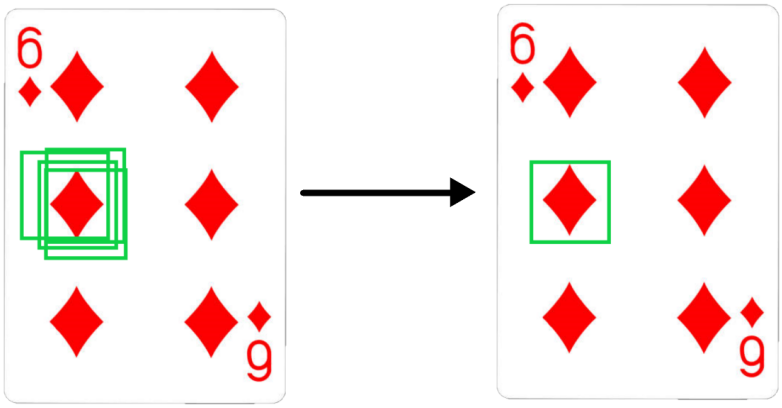

Figura 13: Supresión de no máximos

La función encargada de esta tarea es *nonMaximumSupression()* que recibe como entrada una lista de diccionarios de la forma $\{coordenadas, valor\}$. Para cada elemento $maxima[i]$ de dicha lista se establece que una detección $maxima[j]$ pertenece al mismo objeto que $maxima[i]$ si las coordenadas de estas son al menos un 95 % parecidas. Esto se hace tanto para x como para y, por lo que en el código se definen los rangos:

```
rangeX = (maxima[i]['coords'][0] - 0.05*maxima[i]['coords'][0],
          maxima[i]['coords'][0] + 0.05*maxima[i]['coords'][0])

rangeY = (maxima[i]['coords'][1] - 0.05*maxima[i]['coords'][1],
          maxima[i]['coords'][1] + 0.05*maxima[i]['coords'][1])
```

Una vez que ya se sabe cómo identificar qué detecciones pertenecen al mismo objeto, es necesario decidir cual de estas se conserva. Se decide que se conservará la detección que tenga el valor más alto, por lo que cada elemento $maxima[i]$ comienza con un flag $keep = True$, si un elemento $maxima[j]$ del mismo objeto tiene un valor más alto entonces se cambia el flag de $maxima[i]$ a $keep = False$. Después de recorrer la lista completa para cada elemento $maxima[i]$ se conservan sólo los que tengan el flag $keep = True$.

Aplicando las ideas recién descritas se crea la función *nonMaximumSupression()* que se muestra en el **Listing 11**:

```
def nonMaximumSupression(maxima):
    filtered = []
    for i in range(len(maxima)):
        keep = True
        rangeX = (maxima[i]['coords'][0] - 0.05*maxima[i]['coords'][0],
                  maxima[i]['coords'][0] + 0.05*maxima[i]['coords'][0])
        rangeY = (maxima[i]['coords'][1] - 0.05*maxima[i]['coords'][1],
                  maxima[i]['coords'][1] + 0.05*maxima[i]['coords'][1])
        for j in range(len(maxima)):
            if (maxima[j]['coords'][0] in range(int(rangeX[0]),int(rangeX[1]))) &
               (maxima[j]['coords'][1] in range(int(rangeY[0]),int(rangeY[1]))):
                if maxima[j]['value'] > maxima[i]['value']:
                    keep = False
        if keep:
            filtered.append(maxima[i])
    return filtered
```

Listing 11: Función de supresión de no máximos

4. Resultados

A continuación se presentan las detecciones de los patrones encontrados en las imágenes entregadas por el equipo docente.

4.1. Seis de diamantes

Para la imagen de la carta del seis de diamantes, mostrada en la **Figura 14**:

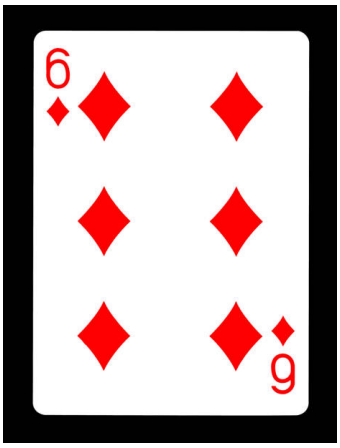

Figura 14: Carta del seis de diamantes

Se deben detectar los seis diamantes grandes presentes en la carta, los dos diamantes pequeños debajo de los 6 no. Aplicando el algoritmo de detección de un patrón general explicado en esta sección, se obtiene el resultado mostrado en la **Figura 15**:

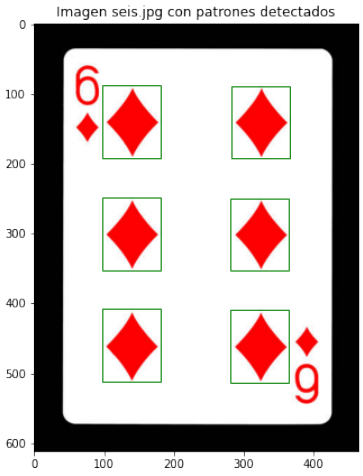

Figura 15: Detección de diamantes en la carta

4.2. Cervantes

Para la imagen que contiene un párrafo del libro "El ingenioso hidalgo don Quijote de la Mancha", mostrada en la **Figura 16**:

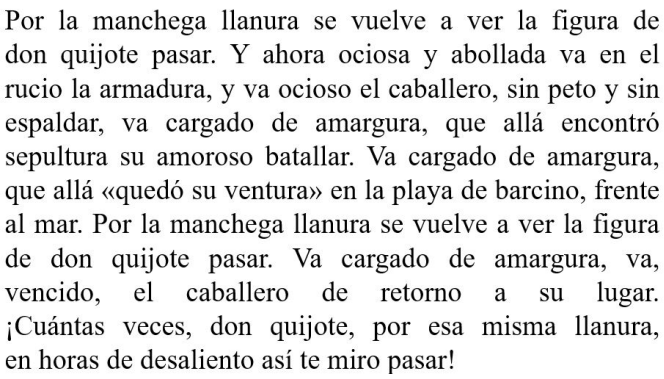

Figura 16: Párrafo del Quijote

Se deben detectar instancias de la palabra "quijote", aplicando el algoritmo de detección de un patrón general se obtiene el resultado mostrado en la **Figura 17**:

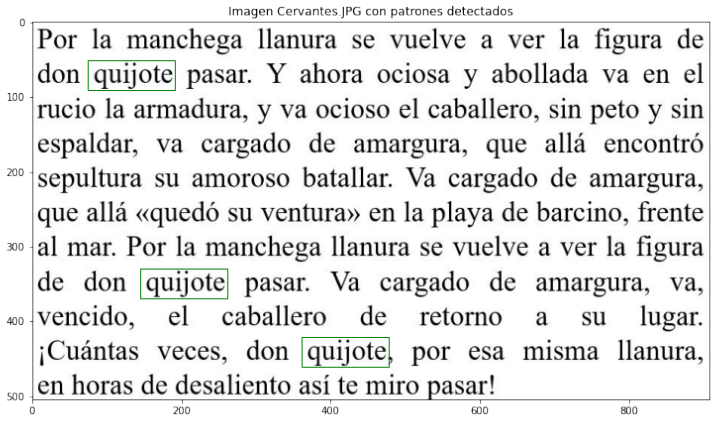

Figura 17: Detección de "quijote'

4.3. Mancha

Para la imagen que contiene el primer párrafo del libro "El ingenioso hidalgo don Quijote de la Mancha", mostrada en la **Figura 18**:

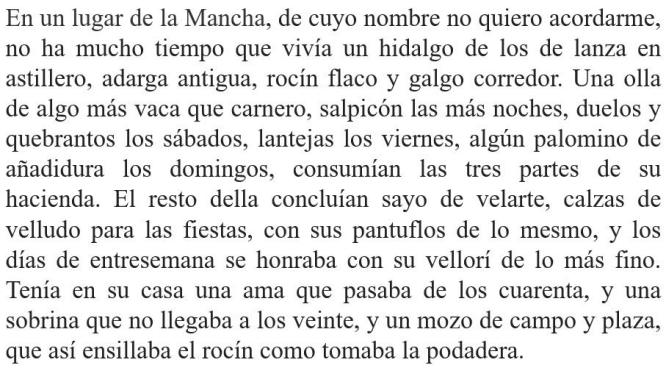
En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda. El resto della concluían sayo de velarte, calzas de velludo para las fiestas, con sus pantuflos de lo mesmo, y los días de entresemana se honraba con su vellorí de lo más fino. Tenía en su casa una ama que pasaba de los cuarenta, y una sobrina que no llegaba a los veinte, y un mozo de campo y plaza, que así ensillaba el rocín como tomaba la podadera.

Figura 18: Primer párrafo del Quijote

Se deben detectar instancias de la palabra "no", aplicando el algoritmo de detección de un patrón general se obtiene el resultado mostrado en la **Figura 19**:

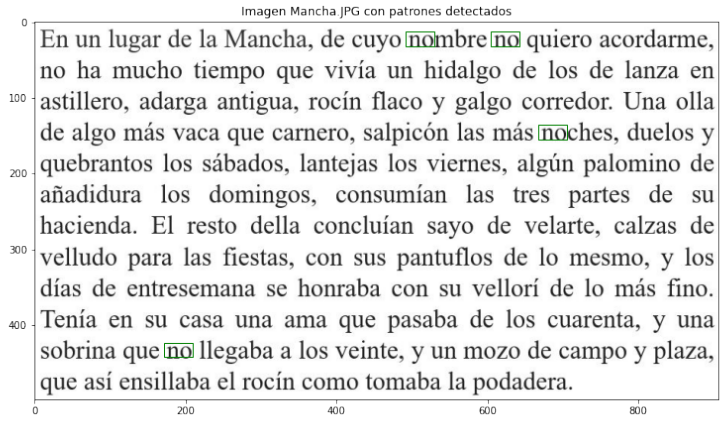

Figura 19: Detección de "no'

4.4. Sopa de letras

Para la imagen que contiene una sopa de letras, mostrada en la **Figura 20**:

Figura 20: Sopa de letras

Se deben detectar instancias de la palabra "CAT" en distribución vertical y de la palabra "DOG" en distribución horizontal, aplicando el algoritmo de detección de un patrón general se obtienen los resultados mostrados en la **Figura 21**:

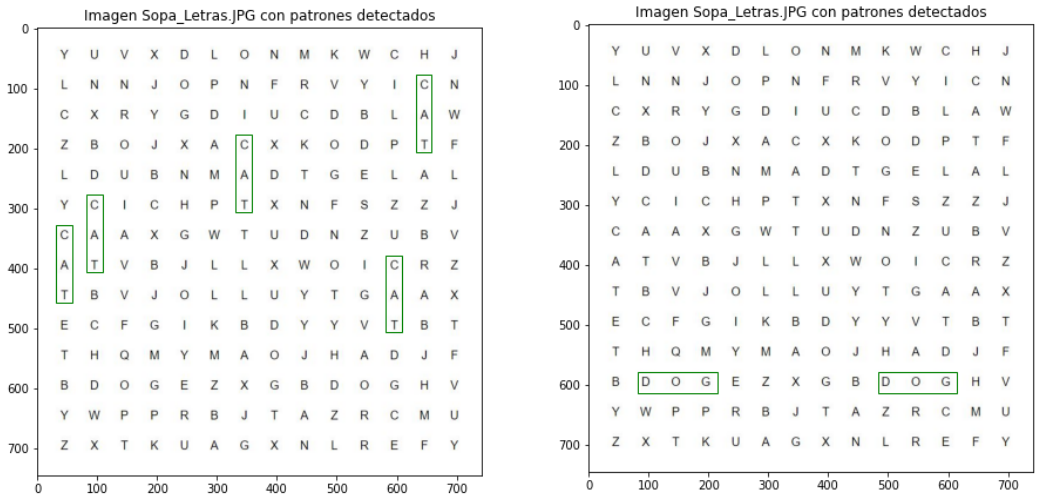

Figura 21: Detección "'CAT" (izquierda) y "DOG" (derecha)

4.5. Tablero de ajedrez

Para la imagen que contiene un tablero de ajedrez, mostrada en la **Figura 22**:

Figura 22: Tablero de ajedrez

Se deben detectar a los caballos y a los peones, aplicando el algoritmo de detección de un patrón general se obtienen los resultados mostrados en las **Figuras 23 y 24**:

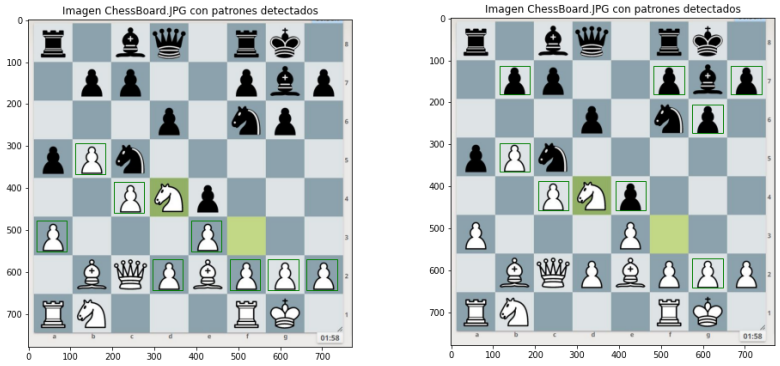

Figura 23: Detección de peones

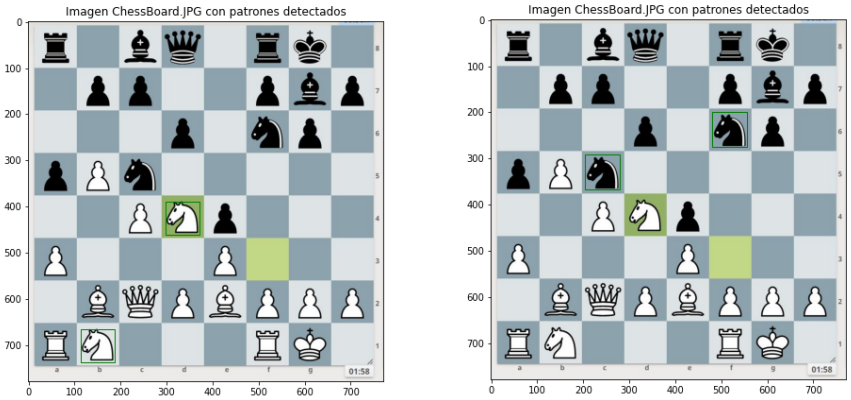

Figura 24: Detección de caballos

4.6. Cartas

Para la imagen que contiene varias cartas de diamantes del naipe inglés, mostrada en la **Figura 25**:

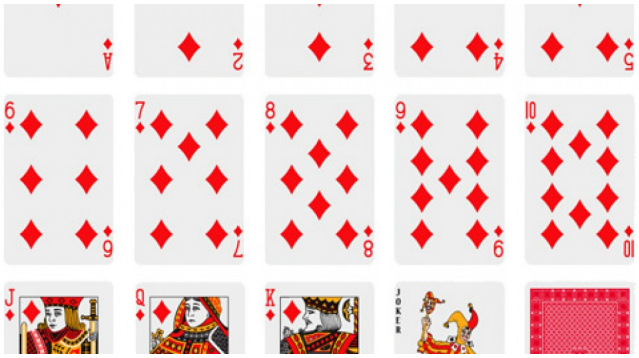

Figura 25: Cartas de diamantes

Se deben detectar los diamantes grandes, al igual que en el caso de la carta del seis de diamantes sola, no se deben detectar los diamantes pequeños debajo de cada número. Aplicando el algoritmo de detección de un patrón general se obtiene el resultado mostrado en la **Figura 26**:

Figura 26: Detección de diamantes en múltiples cartas

5. Análisis de resultados

Se puede apreciar que en todas las imágenes mostradas en la **Sección 4**, se detectan casi todas las instancias del patrón seleccionado. En particular, para el primer párrafo del quijote (**Figura 19**) hay un solo "no" que no fue detectado junto a otras palabras que terminan en no como "fino" que tampoco fueron detectadas. Sin embargo, esto se deba a que si el umbral de detección se baja un poco se detectan todas las instancias de "no" pero también se detectan algunos "mo", como se muestra en la **Figura 27**:

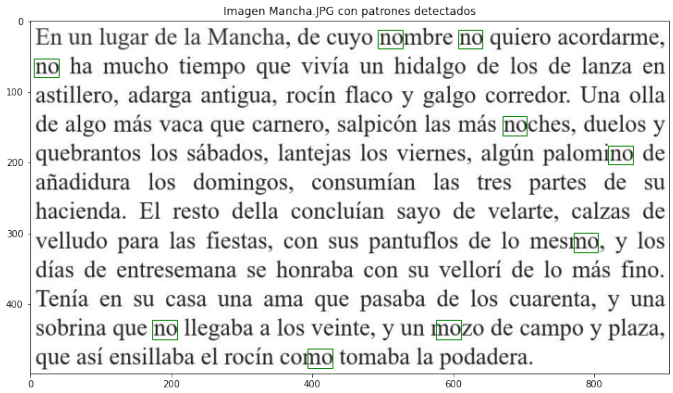

Figura 27: Detección de "no" con menor umbral

Dado que la instrucción sólo pedía la palabra "no", se optó por mantener el umbral con tal de omitir las "mo" que si bien son casi idénticas al patrón pedido, no son exactamente lo que se busca.

Respecto al tablero de ajedrez, se deben separar las detecciones en cuatro casos, los peones blancos y negros, y los caballos blancos y negros. Se puede apreciar en la **Figura 23** que al buscar peones blancos (lado izquierdo), todos estos fueron detectados, mientras que al buscar peones negros (lado derecho), se omitieron unos cuantos peones negros y se incluyeron unos cuantos peones blancos. Esto se debe a que los peones negros que están en casillas negras se vuelven "invisibles" cuando se binariza la imagen, por lo que, con el método implementado, es imposible detectarlos. Por otra parte, en la **Figura 24** se puede apreciar que tanto los caballos blancos como los negros son detectados de manera correcta, esto se debe a que sus dibujos son bien distintos y pueden resaltar en ambos fondos.

Por último, para las cartas de diamantes de la **Figura 25** se puede apreciar que se detectan todos los diamantes grandes, excepto los que están junto a las cartas J, Q y K. Esto se debe a que, si bien son más grandes que los diamantes pequeños de sus respectivas cartas, no son tan grandes como los diamantes de las otras cartas.

Este método tiene como ventajas que es bastante simple de implementar, es intuitivo pues el esquema de Hubel y Wiesel se basa básicamente en cómo vemos los mamíferos, y es bastante rápido si la convolución utilizada es rápida. Mientras que de desvetnajas tiene que no es un detector invariante a la escala, puesto que los objetos detectados tienen que ser básicamente del mismo tamaño que el filtro detector, también no es automático, puesto que hay que escoger manualmente tanto el patrón que se quiere detectar como el umbral de detección (y esto puede llegar a ser bastante complejo), y falla cuando se intentan detectar patrones que tienen un fondo del mismo color (como los peones negros en las casillas negras) pues al binarizar estos "desaparecen".

El problema de la escala se podría solucionar calculando el *state space* tanto para las imágenes de búsqueda como para los filtros detectores, y buscar los patrones en todas las escalas generadas. La automatización es un poco más compleja, no tiene sentido automatizar la parte en la que se escoge el patrón deseado, y se podría automatizar la elección del umbral escogiendo un valor arbitrario como el 80 % del valor máximo, pero esto llevaría a detecciones de menor calidad. Por último para resolver el problema de los patrones que desaparecen con la binarización, se podrían escoger mejores umbrales de binarización, pero esto añade otro paso manual al procedimiento.

6. Conclusión

Se logra implementar en Python todas las funciones necesarias para realizar la detección de patrones de manera interactiva. Se evidencia lo bien que funciona este detector en las condiciones ideales, es decir, se buscan patrones del mismo tamaño en imágenes que no presentan cambios de escala ni rotaciones ni dentro ni fuera del plano.

Se logran detectar los patrones solicitados en las imágenes entregadas en casi todos los casos, a excepción de algunos que ya fueron discutidos en la Sección 5.

Debido a que todas las funciones se programaron en bajo nivel, es decir, utilizando casi exclusivamente las funciones básicas de python y numpy, se logra adquirir un mayor entendimiento sobre el funcionamiento de estos métodos y sobre las dificultades de implementarlos.