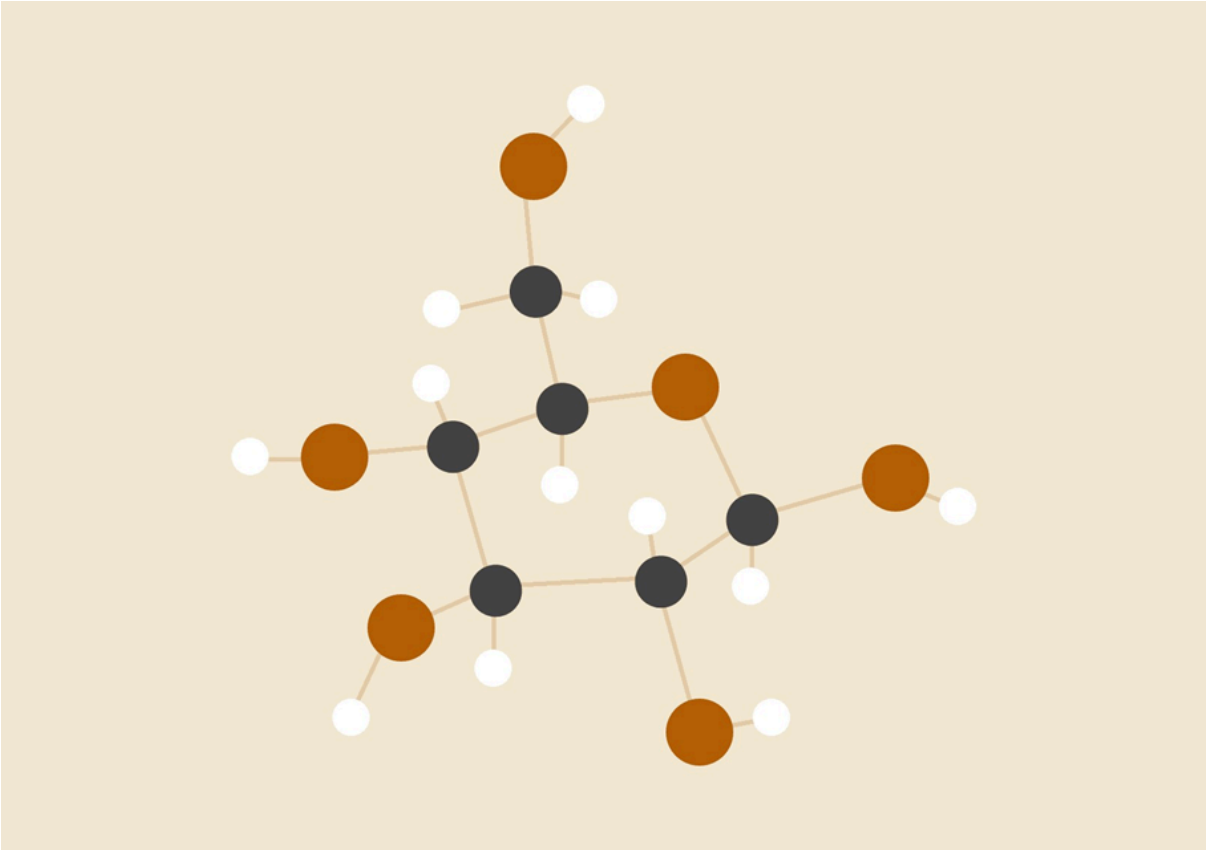
TRABAJO PRÁCTICO 2

IA4.4 Procesamiento Digital de Imágenes
Tecnicatura Universitaria en Inteligencia Artificial



Grupo 06:
Arce, Sofía
Gauto, Lucas
Rizzotto, Camila

11/06/2023
Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

INTRODUCCIÓN

Nuestro trabajo se basa en la resolución de dos ejercicios de procesamiento de imágenes: uno sobre detectar diferentes componentes de una placa electrónica y otro sobre analizar patentes de autos.
Para el primero, aplicamos técnicas como morfología y segmentación para localizar resistencias eléctricas, capacitores electrolíticos y un chip presentes en la placa. Clasificamos los capacitores según su tamaño, y, por último, determinamos la cantidad de resistencias eléctricas que se encuentran en dicha placa.
Para el segundo, tuvimos que analizar patentes que nos fueron dadas, y para lograr esto utilizamos algunas de las técnicas ya nombradas previamente además de otras como detección de bordes y filtrado de componentes conectadas. Terminamos descartando componentes según las dimensiones de las áreas detectadas.

DESCRIPCIÓN DEL ENTORNO DE TRABAJO

Para la realización de estos ejercicios utilizamos el lenguaje Python y generamos un entorno virtual en el cual instalamos y utilizamos los siguientes paquetes:

- Numpy
- Open CV
- Matplotlib

EJERCICIO 1

Archivo: ejercicio 1.py
El objetivo del ejercicio consistió en procesar una imagen de una placa de circuito y clasificar tres tipos principales de componentes electrónicos: resistencias eléctricas, capacitores electrolíticos y un chip. Las tareas específicas incluían la segmentación y distinción visual de estos componentes, la clasificación de los capacitores por tamaño y el conteo de las resistencias eléctricas.

Problemas Enfrentados

Uno de los principales desafíos fue la dificultad de detectar todos los componentes de manera simultánea en la imagen. Para abordar este problema, se optó por tratar cada tipo de componente por separado, aplicando técnicas de procesamiento de imágenes específicas a cada uno. Este enfoque permitió ajustar los métodos de segmentación de manera óptima para cada tipo de componente, logrando así una detección más precisa.

1.A

Se utilizó OpenCV para leer la imagen de la placa y Matplotlib para visualizar tanto la versión en escala de grises como la versión en color.
Nos pareció sencillo generar tres diccionarios diferentes en donde se guarde la información de la ubicación del chip, las resistencias y los capacitores. Así, se nos facilita el apartado b y c de este mismo ejercicio. Para estos diccionarios generamos tres funciones que nos los devuelven, además de mostrarnos por pantalla la imagen de la placa con el componente correspondiente resaltado y señalado. Cada una posee las proporciones físicas consideradas en cada caso.

Comenzamos con la detección del chip: para detectarlo, nos valemos de la función 'detectar_chip' en donde le pasamos una imágen y le aplicamos componentes conectadas. Luego, filtramos por proporciones físicas: la relación largo/ancho y el área. También aplicamos una serie de técnicas de filtrado y transformaciones morfológicas:

- **Eliminación de ruido** mediante un filtro de desenfoque.
- **Detección de bordes** utilizando el operador Sobel.
- **Umbralado** para obtener una imagen binaria de los bordes detectados.
- **Transformaciones morfológicas** (clausura y apertura) para unificar y limpiar las regiones segmentadas.

```
imagenBorrosa = blur = cv2.blur(placa, (21, 21))

gradiente_x = cv2.Sobel(imagenBorrosa, cv2.CV_64F, 1, 0, ksize=3)
gradiente_y = cv2.Sobel(imagenBorrosa, cv2.CV_64F, 0, 1, ksize=3)

magnitud_gradiente = cv2.magnitude(gradiente_x, gradiente_y)

magnitud_gradiente = cv2.convertScaleAbs(magnitud_gradiente)

retval, imagenUmbralada = cv2.threshold(magnitud_gradiente, 20,
255, cv2.THRESH_BINARY)

B = cv2.getStructuringElement(cv2.MORPH_RECT, (10,10))

imgClausura1 = cv2.morphologyEx(imagenUmbralada, cv2.MORPH_CLOSE,
B)

kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (10, 10))

imgAperturada = cv2.morphologyEx(imgClausura1, cv2.MORPH_OPEN,
kernel)

B2 = cv2.getStructuringElement(cv2.MORPH_RECT, (20,20))

imgClausura2 = cv2.morphologyEx(imgAperturada, cv2.MORPH_CLOSE,
B)

kernel = np.array([[0,0,0,0,0],[1,1,1,1,1],[0,0,0,0,0]],np.uint8)

imgDilatada = cv2.dilate(imgClausura2, kernel, iterations=13)

imgErosionada = cv2.erode(imgDilatada, kernel, iterations=15)

kernel = np.array([[0,0,0],[1,1,1],[0,0,0]],np.uint8)

imgErosionada2 = cv2.erode(imgErosionada, kernel, iterations=4)

B2 = cv2.getStructuringElement(cv2.MORPH_RECT, (50,50))
```

```
imgClausura3 = cv2.morphologyEx(imgErosionada2, cv2.MORPH_CLOSE,
B)

#La imagen nos satisface. Identificamos componentes:

chip = detectar_chip(imgClausura3)
```

Seguimos con la detección de resistencias. Se realizó una dilatación en la imagen segmentada previamente para unificar componentes divididos, y la recortamos aprovechándonos de que sabemos dónde comienzan aproximadamente las resistencias para evitar tanta presencia de ruido:

```
imagenParaResistencias = cv2.dilate(imgErosionada2, kernel,
iterations=1)

imagenParaResistenciasRecortada = imagenParaResistencias[900:,:]
```

Este recorte es corregido en la función 'detectar_resistencias', ya que al obtener los stats por componentes conectadas nos dan coordenadas relativas a esta imágen más corta. Simplemente nos guardamos los stats que cumplen con nuestras proporciones físicas sumándole los 900 recortados a la componente y:

```
        if height < width:
            if width <= maxWidth and height <= maxHeight and area >
4000:
                resistencias[f'Resistencia_{contador}'] = componente
                componente_modificado = copy.deepcopy(componente)
                componente_modificado[1] += 900

resistencias_relativas_img_original[f'Resistencia_{contador}'] =
componente_modificado
                contador+=1
```

Para los detección de capacitores, se utilizó una técnica de umbralado para destacar los valores más altos de la imagen, seguido de una clausura y una erosión para segmentarlos. Finalmente, se resaltaron los componentes detectados en la imagen original utilizando colores diferentes para cada tipo de componente:

```
resaltar_componentes_con_diccionario(imgFinal,chip,(255,0,0))
resaltar_componentes_con_diccionario(imgFinal,resistencias,(0,255,0))
resaltar_componentes_con_diccionario(imgFinal,capacitores,(0,0,255))
```

1.B

Clasificamos los capacitores por áreas que se agruparon en tres categorías: chicos, medios y grandes:

```
clasificacion_capacitores = {
    'capacitor_chico':(0,7500),
    'capacitor_medio':(7500,90000),
    'capacitor_grande': (90000,999999)
}
```

Cada tupla indica el área mínima y máxima respectivamente de cada clasificación, así que simplemente recorrimos el diccionario de capacitores, y según el área que tenga lo añadimos a una lista que contiene cada tipo de capacitor. Así, realizando len() a la lista supimos fácilmente cuántos hay de cada uno.

1.C

Finalmente y como expusimos antes, el conteo de resistencias fue rápido por haber creado el diccionario de resistencias:

```
contador_resistencias = 0
for i in resistencias.keys():
    contador_resistencias += 1
print('Las resistencias eléctricas detectadas en total son:
',contador_resistencias)
```

EJERCICIO 2

2.A

Archivo: ejercicio_dos_a.py

Definimos una lista vacía que contendrá las coordenadas de las patentes en las imágenes y que luego la usaremos en el apartado b al igual que la variable img_patentes, que contiene la ruta de las imágenes de las patentes.

Función: ‘`proc_patentes(img)`‘
Argumento: ruta de la imagen original
Devuelve: imagen blureada, imagen con tophat, imagen top hat en escala de grises, y la imagen top hat con cierre

Comenzamos procesando la imagen de la patente original, nos deshacemos del ruido con *blurring* gaussiano para mejorar la detección del área de las patentes.

```
    K_SIZE_GAUSSIAN_BLUR = (1, 19)
    # Blur para deshacernos del ruido y mejorar la detección de bordes
    blur = cv2.GaussianBlur(source_img, K_SIZE_GAUSSIAN_BLUR, 0)
```

A esa imagen blureada, le aplicamos Top Hat para resaltar la parte clara de las patentes, el tamaño que elegimos para el kernel corresponde aproximadamente a las dimensiones de una patente.

```
    filterSize =(17,3)
    K_TOPHAT = cv2.getStructuringElement(cv2.MORPH_RECT, filterSize)
    tophat_img =
cv2.morphologyEx(blur.copy(),cv2.MORPH_BLACKHAT,K_TOPHAT)
```

Y para mejorar las áreas de las patentes aplicamos clausura:

```
    K_CIERRE = cv2.getStructuringElement(cv2.MORPH_RECT, (13, 3))
    cierre = cv2.morphologyEx(img_gray, cv2.MORPH_CLOSE, K_CIERRE)
    light = cv2.threshold(cierre, 0, 255,cv2.THRESH_BINARY |
cv2.THRESH_OTSU)[1]
```

Una vez procesada la imagen del auto que se le visibiliza la patente, chequeamos que todas las áreas donde están las imágenes se hayan detectado bien.

Función: `'marcar_bordes(img_preprocesada, img_original)':`
Argumentos: imagen preprocesada, ruta de la imagen original
Devuelve: imagen con los contornos detectados

```
    img_original = cv2.imread(img_original)

    # Contornos de las formas
    edges =
cv2.findContours(img_preprocesada,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SI
MPLE)[0]
    canvas = np.zeros_like(img_original)
    cv2.drawContours(canvas,edges, -1, (0,255,0), 2)
    return canvas
```

Luego, detectamos las posibles patentes usando `connectedComponentsWithStats` definimos los mínimos y máximos que las componentes detectadas pueden tener para ser consideradas posibles áreas de patentes:

```
    num_labels, labels, stats, centroids =
cv2.connectedComponentsWithStats(imagen_preproc)

    min_ancho = 40
    max_ancho = 100
    min_area = 562
    max_area = 2440
```

Iteramos por cada componente y chequeamos que estén dentro de esos rangos definidos por los mínimos y máximos y el ratio de la figura para descartar áreas que sean verticales o que no correspondan con la figura rectangular horizontal de las patentes:

```
    posibles_patentes=[]

    for contorno in range(1, num_labels):
        x = stats[contorno, cv2.CC_STAT_LEFT]
        y = stats[contorno, cv2.CC_STAT_TOP]
        w = stats[contorno, cv2.CC_STAT_WIDTH]
        h = stats[contorno, cv2.CC_STAT_HEIGHT]
        area = stats[contorno, cv2.CC_STAT_AREA]
        ratio = w/h

        if min_area < area < max_area and 1.739 < ratio < 3.7 and
min_ancho < w < max_ancho:
            posibles_patentes.append([x,y,w,h])
```

En caso de que se haya detectado una sola componente que cumpla con las características, la devolvemos.

```
    if len(posibles_patentes) == 1:
        return posibles_patentes[0]
```

pero, como es posible que en la imagen se detecten areas con las mismas características mas de una vez, elegimos dentro de esas, aquella que tenga mas cantidad de negro en el recorte de la imagen original.

```
    elif len(posibles_patentes) > 1:
        posibles_patentes = contar_pixels_negros(img_orginal,
posibles_patentes)
```

Función: `'contar_pixels_negros(img_orginal, posibles_patentes)'`
Argumentos: imagen preprocesada, lista con las coordenadas de las posibles áreas de las patentes
Devuelve: una lista con las componentes del area que contiene mas negro y que por lo tanto corresponde al de la patente

```
    img_original = cv2.imread(img_original)

    region_patente=[]
    max=0
    for recuadro in coord_de_patentes:
        x=recuadro[0]
        y=recuadro[1]
        w=recuadro[2]
        h=recuadro[3]
```

```
        patente = img_original[y:y+h, x:x+w]

        number_of_white_pix = np.sum(patente == 0)
        if number_of_white_pix > max:
            region_patente=recuadro
```

elemento que se retorna en la función que es llamada:

```
    return region_patente
```

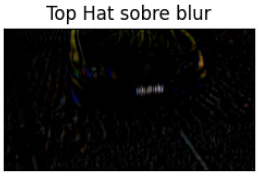
Ahora con las coordenadas de la patente detectada, agregamos ese *array* a la lista “patente” (variable que posteriormente usamos en el ej 2)

```
patentes.append(coord_de_pat)
```

Finalmente, recortamos de nuestra imagen original la patente porque a main no le especificamos que no muestre las imágenes (con plot=false)

Función: `'mostrar_areas_detectadas(img_original, coordenadas)'`
Argumentos: imagen original, lista con las coordenadas del área de la patente
Devuelve: la imagen de la patente recortada de la original

Se muestra en pantalla todas las imágenes por las etapas de procesado y filtrado hasta conseguir la patente:



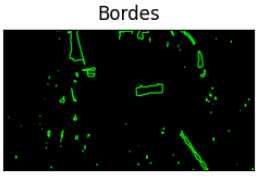


Imagen Original

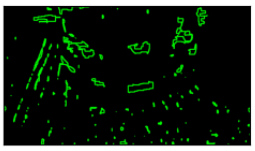
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

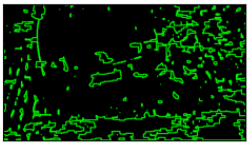
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

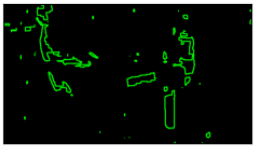
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

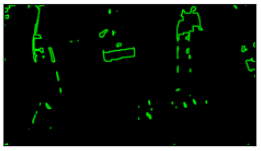
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

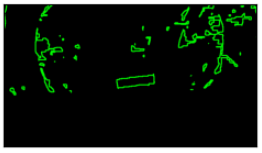
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

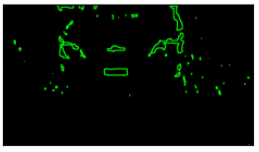
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

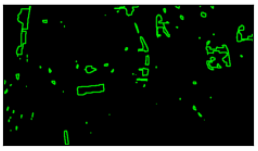
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

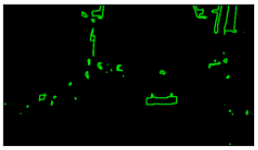
Original con blurring

Top Hat sobre blur

Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente

Imagen Original

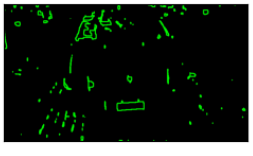
Original con blurring

Top Hat sobre blur

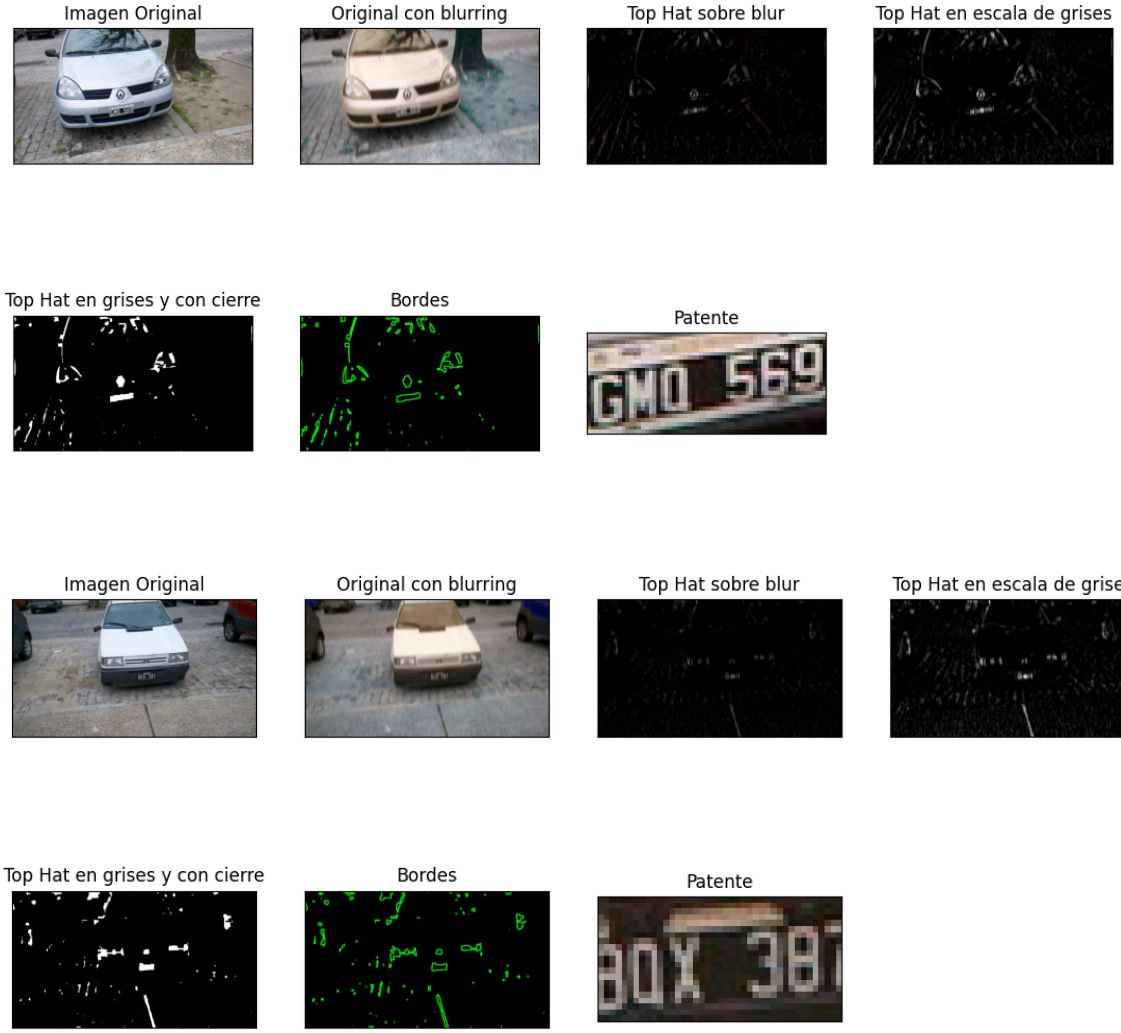
Top Hat en escala de grises

Top Hat en grises y con cierre

Bordes



Patente



2.B

Archivo: ejercicio_dos_b.py

Reutilizamos main con plot=False y la ruta de las imágenes del apartado a:

```
from ejercicio_dos_a import main, img_patentes
```

Con las imágenes de las patentes ya segmentadas, la convertimos en escala de grises, umbralizamos y segmentamos las letras de la patente siguiendo un procedimiento similar al del ejercicio A usando `connectedComponentsWithStats` y aplicando un filtro de ratio para quedarnos con aquellas que se correspondiera con las de las letras.

```
for i in range(0,len(img_patentes)):

    imagen_original = cv2.imread(img_patentes[i])
    x=patentes[i][0]
    y=patentes[i][1]
```

```
    w=patentes[i][2]
    h=patentes[i][3]

    patente = imagen_original[y:y+h, x:x+w]

    # Convierto a escala de grises
    img_gray_patente = cv2.cvtColor(patente, cv2.COLOR_BGR2GRAY)

    # Binarizo
    umbralizada = cv2.threshold(img_gray_patente, 0,
255,cv2.THRESH_BINARY | cv2.THRESH_OTSU)[1]

    num_labels, labels, stats, centroids =
cv2.connectedComponentsWithStats(umbralizada)

    contornos_de_interes = []

    posibles_patentes=[]

    for contorno in range(1, num_labels):
        x = stats[contorno, cv2.CC_STAT_LEFT]
        y = stats[contorno, cv2.CC_STAT_TOP]
        w = stats[contorno, cv2.CC_STAT_WIDTH]
        h = stats[contorno, cv2.CC_STAT_HEIGHT]
        area = stats[contorno, cv2.CC_STAT_AREA]
        ratio = w/h

        if  h/w > 1:
            contornos_de_interes.append([x,y,w,h])
```

Encuadramos las áreas de que corresponden a las letras de las patentes con

```
    for cnt in contornos_de_interes:
        x, y, w, h = cnt
        cv2.rectangle(patente, (x, y), (x + w, y + h), (0, 255, 0), 1)

    plt.imshow(patente, cmap='gray'), plt.show()
```

Nuestro script detecta componentes de letras en 11/12 imágenes.

CONCLUSIONES

Tanto en la detección de componentes electrónicos en una placa como en la segmentación de letras en patentes, enfrentamos problemas similares: no se detectaban todos los elementos de forma sencilla. Para superar esto, tratamos cada tipo de componente o carácter por separado, ajustando los tratamientos de imagen y las características físicas o proporciones que utilizamos para filtrar lo que se detectaba según las necesidades específicas de cada caso. Aún así, notamos que con las herramientas que disponemos no pudimos llegar a segmentar todo lo requerido, como fue en el último ejercicio.
Entendemos, por los resultados de este TP, que un enfoque iterativo y flexible es imprescindible y fue el que permitió alcanzar la mayoría de los objetivos de segmentación y clasificación que se nos propusieron, demostrando la importancia de adaptar las técnicas de procesamiento de imágenes a las características particulares de cada tarea.