

Detector de Sudokus Automático

Práctica OpenCV

Esther Cuervo Fernández
16 de enero de 2019

SISTEMAS HARDWARE Y SOFTWARE DE CAPTURA Y
VISUALIZACIÓN DE IMAGEN

Máster en Ingeniería Informática
UNIVERSIDAD DE VALLADOLID

Índice

1. Introducción	2
2. Programa elaborado	2
2.1. Estructura del código	2
2.2. Ajuste del puzzle	3
2.2.1. Lectura y preparado de la imagen	5
2.2.2. Detección de contornos	6
2.2.3. Detección y ajuste del puzzle	8
2.2.4. Resultados	11
2.3. Extracción y reconocimiento de números	14
2.3.1. Detector de números	18
2.3.2. Modelo	20
2.4. Utilización del pipeline	24
3. Utilización del programa	25
4. Resultados y conclusiones	26

1. Introducción

Este documento describe la práctica final elaborada para la asignatura Sistemas Hardware y Software de Captura y Visualización de Imagen del primer curso del Máster en Ingeniería Informática.

El objetivo de dicha práctica es desarrollar un sistema que permita detectar sudokus a partir de imágenes tomadas en un entorno no controlado, es decir, que puedan contener rotaciones, otros hechos, etc.

Para ello se han utilizado técnicas de visión por ordenador, así como una red neuronal para clasificado de dígitos.

2. Programa elaborado

En la siguiente sección se explica de manera detallada la técnica y código utilizado para detectar sudokus automáticamente, así como la justificación de su uso.

2.1. Estructura del código

El código se ha programado en su totalidad en Python3, y utiliza las siguientes librerías:

- OpenCV
- Pillow
- Tensorflow
- Numpy

Se ha implementado todo el código mediante el paquete `autosudoku`, con la siguiente jerarquía:

```

autosudoku
├── classifiers
│   ├── models
│   │   └── char74k
│   │       ├── variables
│   │       └── saved_model.pb
│   └── numberclassifier.py
├── detectors
│   ├── contourdetector.py
│   └── rectangledetector.py
└── model
    ├── chars74k.py
    ├── cnn.py
    ├── main__.py
    ├── pipe.py
    └── show.py
    └── UbuntuMono-B.ttf

```

De este modelo jerárquico se han excluido los ficheros `__init__.py`, necesarios para que Python considere los ficheros dentro de un directorio parte de un paquete u módulo.

Por tanto el paquete `autosudoku` se compone de los siguientes módulos:

- `classifiers`. Este módulo gestiona la clasificación de números dada una imagen del mismo.
- `detectors`. Este módulo contiene un detector de contornos, que realiza un umbralizado y detección de contornos a una imagen dada, y un detector de rectángulos, que toma dichos contornos y devuelve el polinomio cerrado de grado 4 más grande encontrado entre los mismos.
- `model`. Este módulo contiene una red neuronal para clasificación de números.

2.2. Ajuste del puzzle

Este es el primer paso en la pipeline de trabajo. Las imágenes que se introducen al programa son fotos de sudokus, conteniendo muchas veces otros elementos. Esta imagen es un ejemplo del tipo de imágenes que procesará el algoritmo:

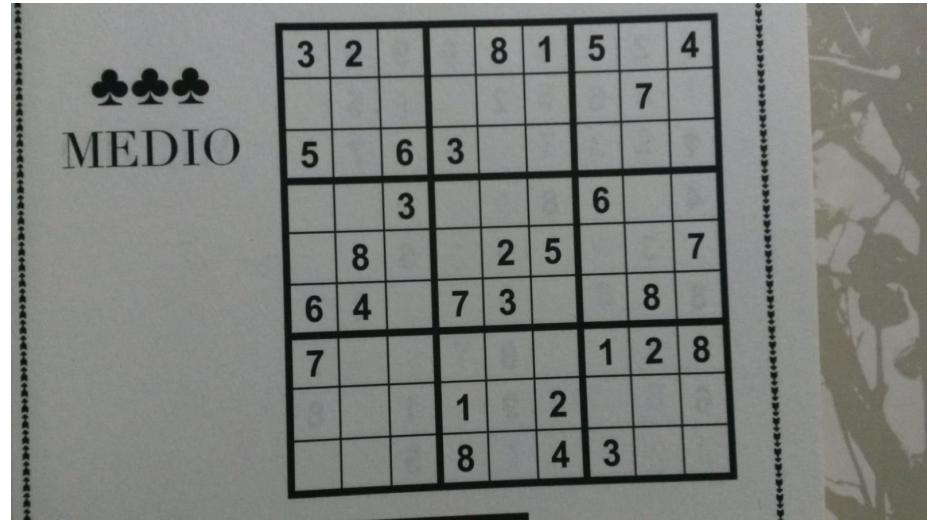


Figura 1: Imagen de ejemplo introducida al programa

Nuestra intención es extraer sólo el puzzle, ignorando otros hechos como pueden ser el fondo, las letras, o cualquier otro dibujo incluido en la imagen. A continuación se muestra un ejemplo del resultado deseado de esta etapa:

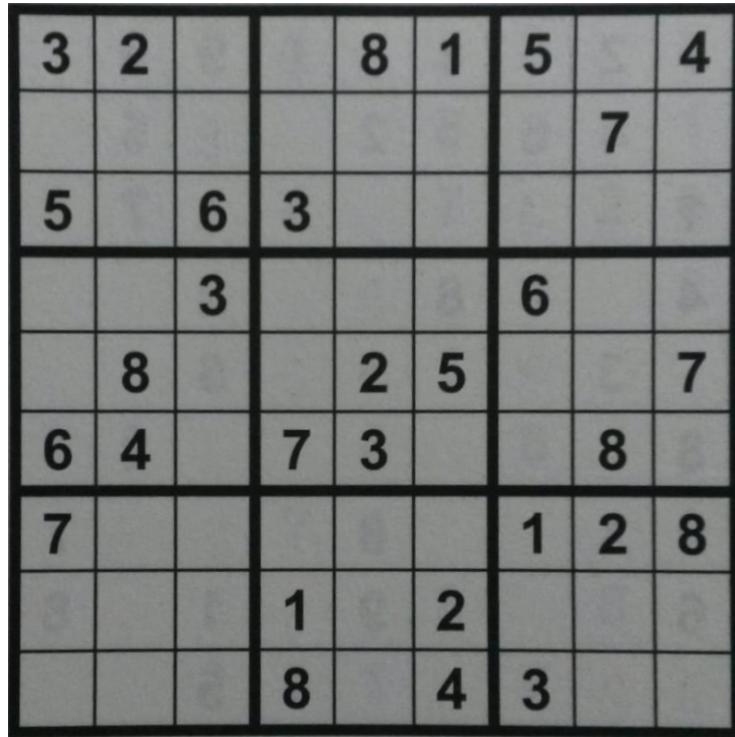


Figura 2: Resultado del ajuste del puzzle en la imagen de ejemplo

2.2.1. Lectura y preparado de la imagen

La imagen es leída en modo BGR y almacenada como una matriz `numpy`.

Tras ello convertimos la imagen a escala de grises, y aplicamos un leve filtro de emborronamiento Gaussiano, que ayudará a reducir el ruido previo a la segmentación de la imagen.

```
image = cv2.imread(filename)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
gaussian = cv2.GaussianBlur(gray,(5,5),0)
```

Código Fuente 1: Código utilizado para realizar la lectura y preparado de la imagen

El único filtro con parámetros en esta etapa es el emborronamiento Gaussiano, que toma el tamaño del kernel y sigmaX (desviación estándar en X).

El valor 0 en sigmaX implica que este valor se calculará a partir del tamaño del kernel, por tanto el único parámetro a ajustar es el tamaño de la ventana (kernel).

Según aumenta este parámetro aumenta el nivel de emborronamiento, por lo que un valor demasiado grande causaría un emborronamiento demasiado alto, ya que simplemente se busca reducir el ruido que pueda haber en la imagen.

2.2.2. Detección de contornos

El siguiente paso es detectar los contornos de la imagen. Hay dos aproximaciones principales a la hora de conseguir contornos:

- **Umbralizado.** Proceso por el cual se binariza una imagen dividiendo su histograma en dos respecto a un umbral, sea estático o dinámico.
- **Detección de bordes.** Proceso por el cual se detectan cambios en la función de intensidad, correspondientes a bordes.

Para la primera aproximación se ha utilizado un umbralizado, más específicamente el método de Otsu, ya que este método funciona bien con imágenes cuyo histograma es bimodal.

Una imagen bimodal es aquella en la cual el histograma tiene dos picos bien definidos. Otsu es capaz entonces de calcular el umbral y dividir la imagen con resultados muy satisfactorios.

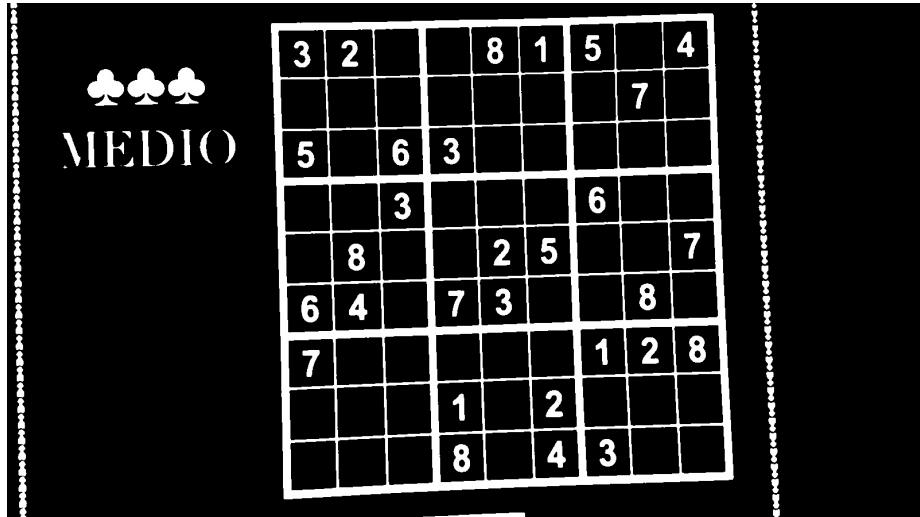
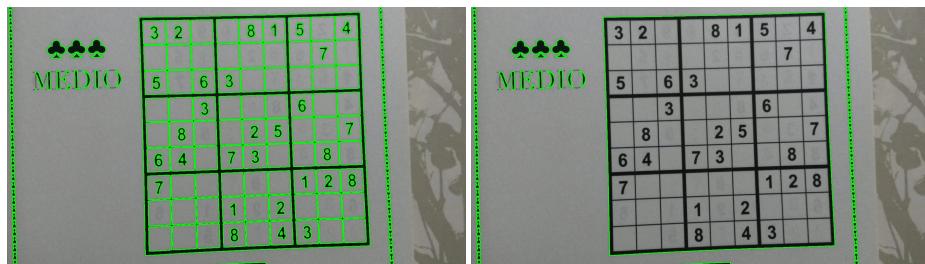


Figura 3: Resultado de aplicar el método Otsu de umbralizado adaptativo

La implementación del algoritmo de búsqueda de contornos nos permite quedarnos con todos los contornos encontrados, o solamente aquellos más arriba en el árbol jerárquico de contornos. Se puede ver la diferencia entre ambas opciones en la siguiente imagen:



(a) Opción RETR_TREE

(b) Opción RETR_EXTERNAL

Para esta primera etapa podemos utilizar la segunda opción, ya que solo nos interesa el contorno exterior del sudoku.

Por tanto el resultado de esta fase es una lista con tantos elementos como contornos cerrados se hayan encontrado. Cada contorno está formado por una lista de puntos.

```

_,threshold = cv2.threshold(gaussian,0,255,
                             cv2.THRESH_BINARY+cv2.THRESH_OTSU)
threshold = cv2.bitwise_not(threshold)

_, contours = cv2.findContours(threshold, cv2.RETR_EXTERNAL,
                               cv2.CHAIN_APPROX_SIMPLE)

```

Código Fuente 2: Código utilizado para realizar la detección de contornos

2.2.3. Detección y ajuste del puzzle

La siguiente etapa de la pipeline es conseguir detectar cuál de los contornos encontrados pertenece al puzzle.

La aproximación que se ha llevado a cabo está parcialmente basada en el tutorial de pyimagesearch “OpenCV shape detection” [1].

El primer paso en esta etapa es aproximar cada contorno encontrado con un polinomio cerrado. Esto permite simplificar los contornos, tras lo cual podemos saber qué contornos se han aproximado con grado 4, entre los cuales estará nuestro puzzle.

```

rectangles = []
dist_allowed = 0.04
while(len(rectangles)==0):
    for c in cont:
        peri = cv2.arcLength(c,True)
        approx = cv2.approxPolyDP(c,dist_allowed*peri,True)
        if(len(approx)==4):
            rectangles.append(approx)
    dist_allowed += 0.01
    if(dist_allowed == 1):
        break

```

Código Fuente 3: Fragmento de código que realiza la detección de polinomios cerrados de grado 4

El código anterior trata de aproximar cada contorno con un polinomio mediante `cv2.approxPolyDP(c,epsilon,closed)`. Toma dos parámetros, `epsilon` y `closed`. El primero cambia la calidad de la aproximación, representando la distancia máxima entre el contorno real y la aproximación. El segundo marca si la aproximación es una curva cerrada (el primer y último vértice coinciden) o no.

Para el cálculo de épsilon se utiliza un valor que va aumentando en un 1% mientras no se encuentre ningún polinomio de grado 4 en la imagen. El valor inicial de 4% se obtiene del tutorial mencionado anteriormente [1]. Por otra parte, está claro que queremos una aproximación cerrada.

Tras obtener las aproximaciones nos quedamos con aquellas cuya longitud, es decir número de vértices, es 4 que corresponderán a polinomios de grado 4.

Una vez obtenidos los polinomios de grado 4 en la imagen, podemos ver que, por lo general, pueden haberse aproximado varios contornos.

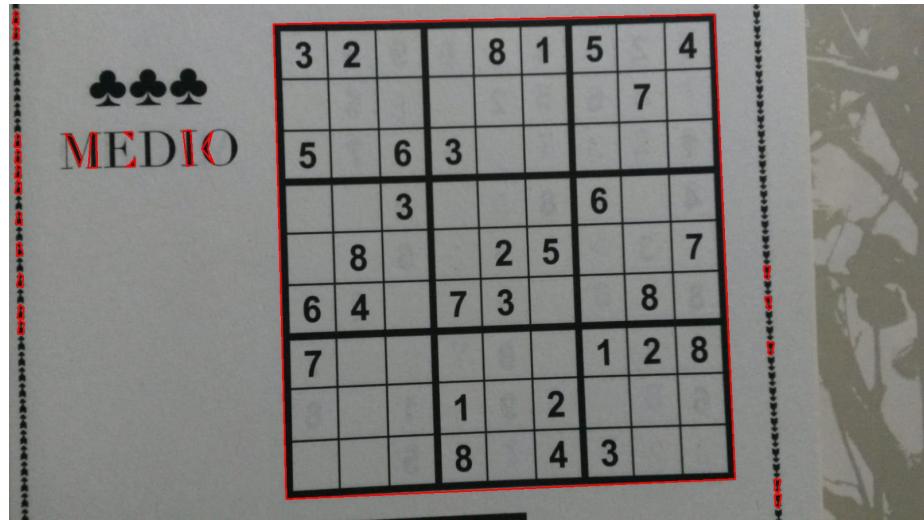


Figura 5: Resultado de la detección de polinomios cerrados de grado 4 dibujado sobre la imagen

Aunque puede, en ciertos casos, no ser así, se ha seleccionado como puzzle el polinomio de área más grande de los encontrados de grado 4.

```
areas = [cv2.contourArea(rect) for rect in rectangles]
biggest_figure_index = [i for i,a in enumerate(areas)
                        if a==max(areas)][0]
biggest_figure = rectangles[biggest_figure_index]
```

Código Fuente 4: Búsqueda del polinomio de área más grande

Tras esto obtenemos los cuatro vértices que forman el puzzle, y podemos realizar una transformación de perspectiva desde dichos cuatro puntos a las cuatro esquinas de una nueva imagen, que sólo tendrá el puzzle.

Para ello debemos asegurar que los puntos están ordenados, para que la relación entre los puntos fuente y destino sea la correcta.

```
xSorted = biggest_figure[np.argsort(biggest_figure[:,0]),:]
lefts = xSorted[:2,:]
rights = xSorted[2:,:]

top_left, bottom_left = lefts[np.argsort(lefts[:,1]),:]
top_right, bottom_right = rights[np.argsort(rights[:,1]),:]

src = np.array([top_left, top_right, bottom_right, bottom_left],
              dtype=np.float32)
```

Código Fuente 5: Fragmento de código que ordena los puntos según las agujas del reloj

Tras esto ya podemos calcular el tamaño de la nueva imagen, para lo cual utilizamos los puntos ordenados, calculando las distancias máximas en el eje X e Y para obtener la anchura y altura respectivamente.

```
new_width = max(abs(src[1,0]-src[0,0]),abs(src[2,0]-src[3,0]))
new_height = max(abs(src[0,1]-src[3,1]),abs(src[1,1]-src[2,1]))
```

Código Fuente 6: Fragmento de código que calcula las nuevas dimensiones de la imagen

Ya podemos obtener la matriz necesaria para transformar la imagen, y realizar la transformación.

```
dst = np.float32([[0,0],[new_width,0],
                  [new_width,new_height],[0,new_height]])
M = cv2.getPerspectiveTransform(src,dst)
result = cv2.warpPerspective(image, M, (new_width, new_height))
```

Código Fuente 7: Fragmento de código que obtiene la matriz y transforma la imagen

Esta operación tiene como resultado la imagen mostrada en la Figura 2.

2.2.4. Resultados

Se ha utilizado un conjunto de 111 imágenes tomadas de dos libros distintos de Sudokus, con dos diseños distintos, el visto en la imagen de muestra de la Figura 1, y uno que cuenta con un borde exterior menos marcado, similar al mostrado en la siguiente imagen:

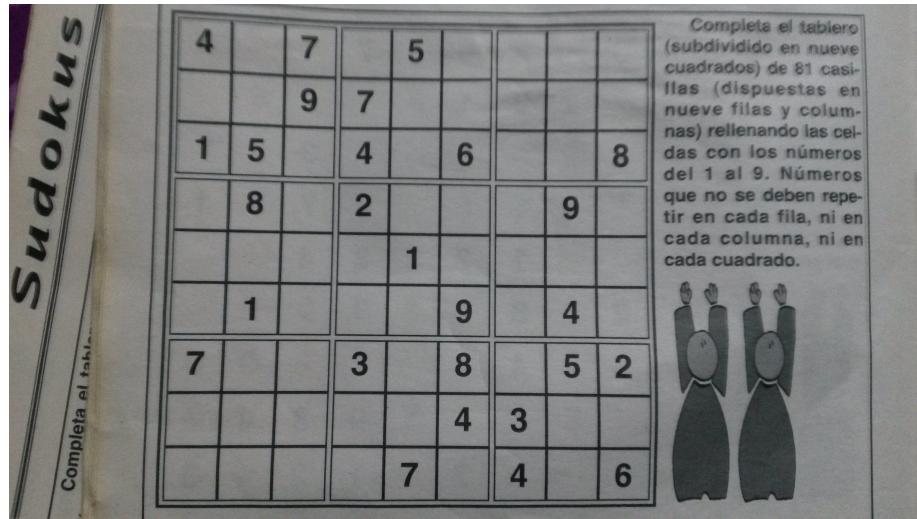


Figura 6: Segundo modelo de imágenes utilizadas

Se ha utilizado esta primera fase del pipeline en todas las imágenes, detectando un correcto ajuste de 86 de estas imágenes, con 25 de ellas resultando en ajustes incorrectos. Esto implica un 77,48 % de aciertos.

La mayoría obtienen uno de los cuadrados que forman parte del puzzle:

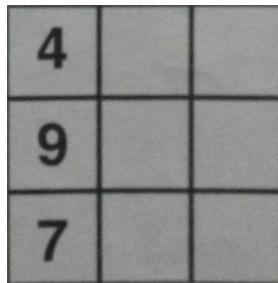
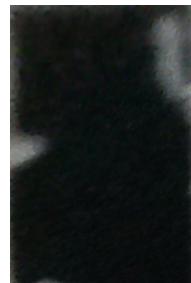


Figura 7: Uno de los resultados incorrectos del algoritmo

Y en otros casos menos comunes, se detectan pequeños segmentos o manchas:



(a) Pequeño segmento resultado del algoritmo



(b) Mancha como resultado del algoritmo

Una observación inicial de estas 25 imágenes con malos resultados nos permite observar que la mayoría se trata del segundo tipo de imagen, observable en la Figura 6, muchas veces mal encuadradas, tal que el borde exterior del sudoku no es visible en su totalidad. Estas imágenes no podrán ser aproximadas por nuestro algoritmo, ya que es necesario que pueda detectar el contorno en su totalidad.

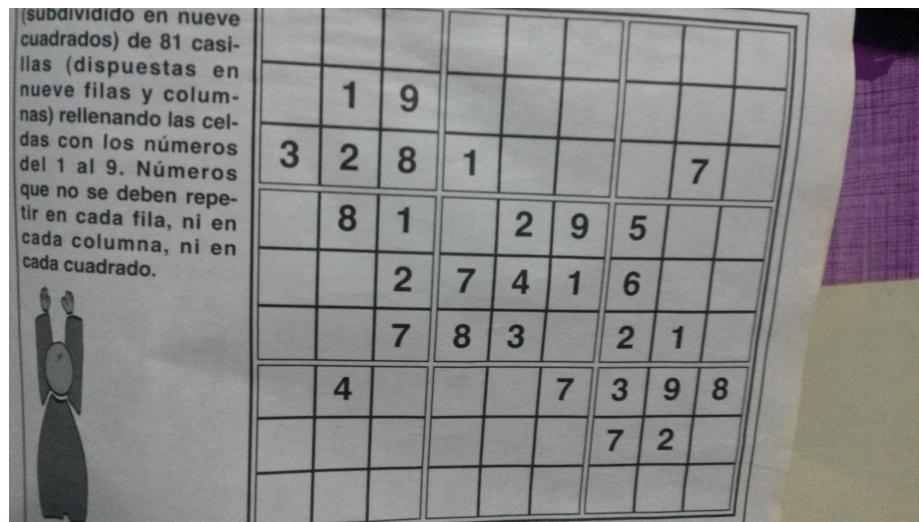


Figura 9: Imagen de ejemplo en la cual el borde exterior no es visible en su totalidad

Después de examinar los resultados intermedios del algoritmo para algunas de las imágenes, se ha visto que el thresholding, mientras que funciona muy bien para aquellas imágenes claramente bimodales, no da buenos resultados para otras, como se puede observar en la siguiente imagen:

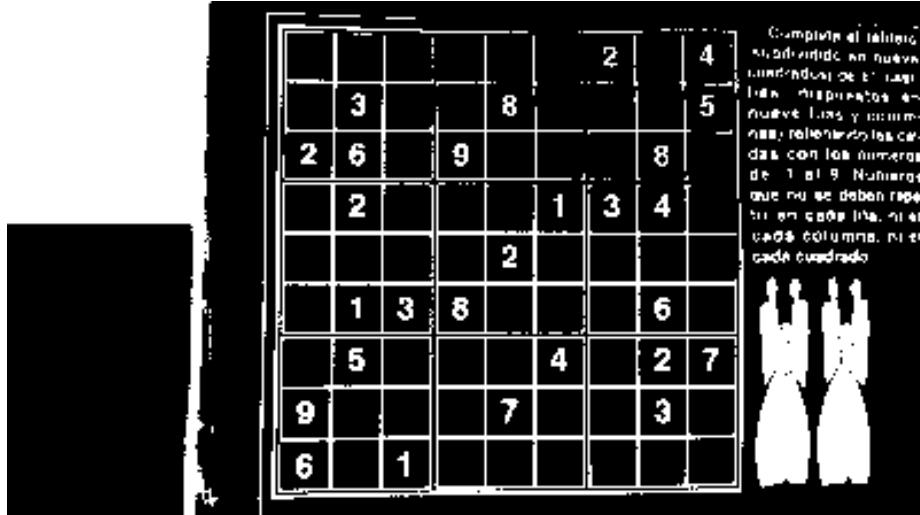
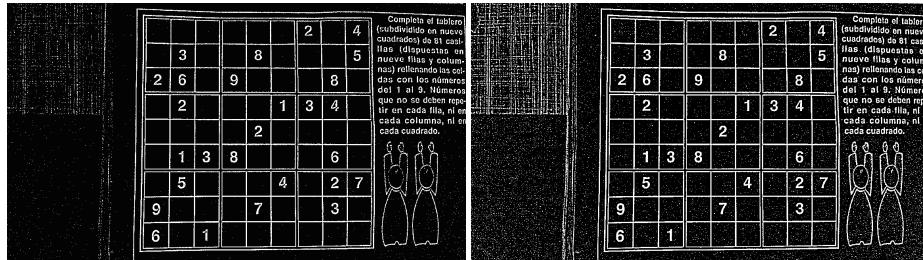


Figura 10: Resultado del método Otsu de thresholding en una de las imágenes con resultado erróneo

Como se puede ver, se pierden algunas de las líneas necesarias para obtener el contorno. Por tanto se ha decidido cambiar el método Otsu por otro tipo de thresholding adaptativo que funcione mejor en imágenes no bimodales, probando con dos: el Gaussiano y el de Medias.



(a) Resultado del umbralizado gaussiano

(b) Resultado del umbralizado de medias

Como se puede observar, el umbralizado de medias tiene más ruido, que aunque podría repararse mediante una operación morfológica, es indeseable en cualquier caso. Por tanto seleccionamos el método gaussiano.

```

threshold = cv2.adaptiveThreshold(gaussian,255,
                                  cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                  cv2.THRESH_BINARY,11,2)

```

Código Fuente 8: Código para realizar umbralizado gaussiano

Tras este cambio volvemos a utilizar la pipeline para ver cuantas imágenes se ajustan correctamente con el nuevo método.

Aplicado sobre las imágenes anteriormente mal ajustadas resulta en 4 de las 25 obteniendo buenos resultados. Sin embargo este resultado no se puede generalizar, ya que 65 de las 86 imágenes bien ajustadas mediante el método de Otsu tienen un mal resultado con el nuevo método.

Uniendo ambos métodos (utilizando Otsu y Gaussian cuando el anterior falle) podemos obtener 90 imágenes bien ajustadas de 111, resultando en un acierto del algoritmo del 81 %.

2.3. Extracción y reconocimiento de números

Partiendo del resultado del paso anterior, es posible volver a realizar la detección de contornos, esta vez obteniendo todos los contornos, utilizando la opción `cv2.RETR_TREE` para obtener el árbol jerárquico de contornos.

Esta jerarquía se trata de un vector contenido 4 valores para cada contorno, [Siguiente, Anterior, Primer Hijo, Padre]. Siguiente y Anterior se refieren a otros contornos en el mismo nivel jerárquico al actual, mientras que Primer Hijo es el primer contorno un nivel por debajo (contenido por el contorno actual). Padre es el contorno que contiene al actual. Si no existe alguno de los valores, aparecen representados con un -1.

De esta manera es posible, en una imagen en la cual los contornos son visibles en su completitud, y correctamente unidos para formar formas cerradas, la separación de casillas vacías, casillas que contienen números, y dichos números. Un ejemplo del resultado de dicha separación es el siguiente dibujado de contornos sobre la imagen resultado del ajuste del puzzle:

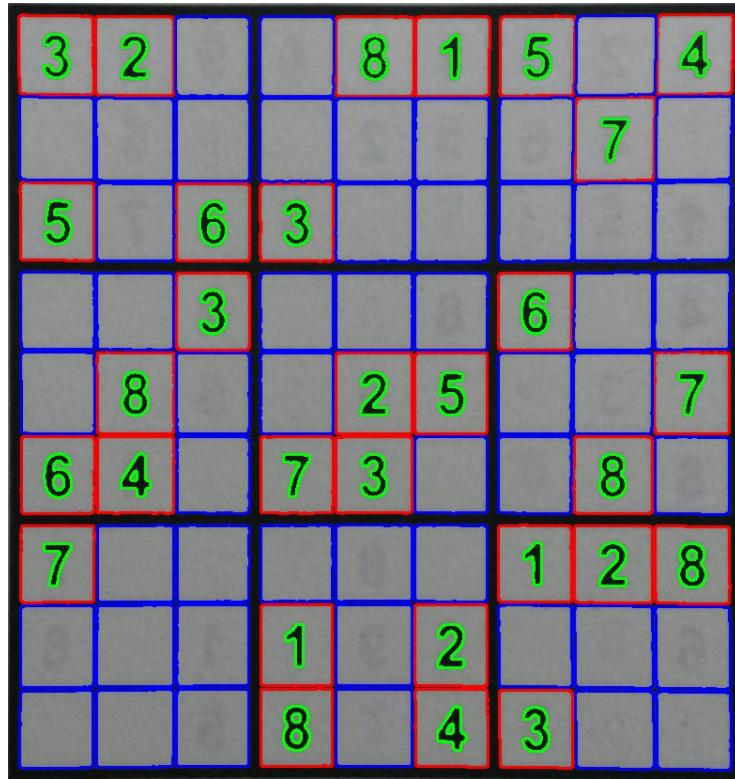


Figura 12: Separación de contornos en casillas vacías (azul), casillas que contienen números (rojo) y números (verde)

El siguiente código realiza esa separación y almacena el contorno correspondiente al número contenido en cada casilla para su uso posterior:

```

_,cont, hierarchy = cv2.findContours(threshold, cv2.RETR_TREE,
                                      cv2.CHAIN_APPROX_SIMPLE)

empty = list()
contain_number = list()
number_contained = list()

for index,(_,_,child,parent) in enumerate(hierarchy[0]):
    if(parent==0):
        if(child!=-1):
            contain_number.append(contours[index])
            number_contained.append(child)
        else:
            empty.append(contours[index])

```

Código Fuente 9: Código que permite separar los contornos según los criterios impuestos

Es importante remarcar que en caso de no encontrar ningún contorno que se adapte a las restricciones dadas, se lanzará una excepción que debe ser gestionada en el script que utilice esta pipeline.

Estos contornos no son detectados en un orden en particular, por lo que el siguiente paso es obtener los vértices de las casillas, tanto vacías como aquellas que contienen números, lo cual nos permitirá ordenar los resultados por filas y columnas, y así obtener una matriz correcta.

Esto se puede conseguir de nuevo aproximando polinomios cerrados, esta vez sin importar el grado, utilizando aquellos contornos en las listas `empty` y `contain_number` y utilizando dichas aproximaciones, más simples, como vértices mediante los cuales ordenar las casillas.

Esto nos proporciona contornos con un número de vértices menor, que ya podemos ordenar de manera similar a como hacíamos en la sección anterior. Para ello obtenemos la mínima coordenada X e Y en cada conjunto de vértices, lo cual nos proporciona la esquina superior-izquierda.

```

mins = list()
all_squares = empty+contain_number
for verts in all_squares:
    xMin = np.amin(verts[:,0,0])
    yMin = np.amin(verts[:,0,1])
    mins.append([xMin,yMin])

```

Código Fuente 10: Código que permite obtener la coordenada superior-izquierda de cada casilla detectada

Podemos ordenar los contornos por filas, ordenando por la coordenada Y de los mínimos, y tras ello iterar por cada fila, ordenando los contornos en dicha fila por columna, ordenando por la coordenada X:

```

sorted_indices = np.argsort(mins[:,1])
# asumimos una matriz cuadrada y que todas las casillas han sido encontradas
matrix_size = int(sqrt(len(all_squares)))

sudoku = np.zeros((matrix_size, matrix_size), dtype=np.int)
for i in range(0,matrix_size):
    indices_row = sorted_indices[i*matrix_size:(i+1)*matrix_size]
    # obtenemos los índices de los contornos de la fila ordenados
    # por la coordenada X de su vértice superior-izquierdo
    indices_row = indices_row[np.argsort(mins[indices_row,0])]

```

Código Fuente 11: Código que permite obtener la matriz de conjuntos correspondiente al puzzle

Dentro de este mismo bucle designamos las casillas vacías con un valor especial (-1) y para el resto, obtenemos el contorno correspondiente al número que contiene.

Para cada contorno correspondiente a un número guardamos en un diccionario: la fila, la columna y el contorno. Hacemos esto para poder llamar a la función `__getNumbers__` con todos los contornos.

La razón por la que se hace esto es porque esta función desencadena la creación de una sesión de Tensorflow, en la cual clasificaremos todos los números obtenidos, en lugar de crear y destruir la sesión por cada número.

```

number_contours = {'i':[],'j':[],'c':[]}
for i in range(0,matrix_size):
    ...
    # si el índice está en el array empty, le asignamos valor -1
    for j,r in enumerate(indices_row):
        if r < len(empty):
            sudoku[i,j] = -1
        else:
            # el array number_contained contiene el índice del contorno
            # que representa el número contenido dentro de cada casilla no-vacía
            c = contours[number_contained[r-len(empty)]]
            number_contours['i'].append(i)
            number_contours['j'].append(j)
            number_contours['c'].append(c)

numbers = __getNumbers__(number_contours['c'])
#colocamos cada número en su lugar
for idx,row in enumerate(number_contours['i']):
    i = row
    j = number_contours['j'][idx]
    n = numbers[idx]
    sudoku[i,j] = n

```

Código Fuente 12: Código que asigna los valores a cada casilla

Tras esto obtendríamos la matriz que representa el sudoku.

2.3.1. Detector de números

La función `__getNumbers__(c)` toma como atributo un array de contornos, y devuelve los números detectados a partir de ellos.

Para ello realiza tres pasos, el primero es el trazado de un rectángulo que contiene a cada contorno mediante la función `cv2.boundingRect(c)`. Esta función devuelve las coordenadas superior-izquierda del rectángulo, su anchura y su altura.

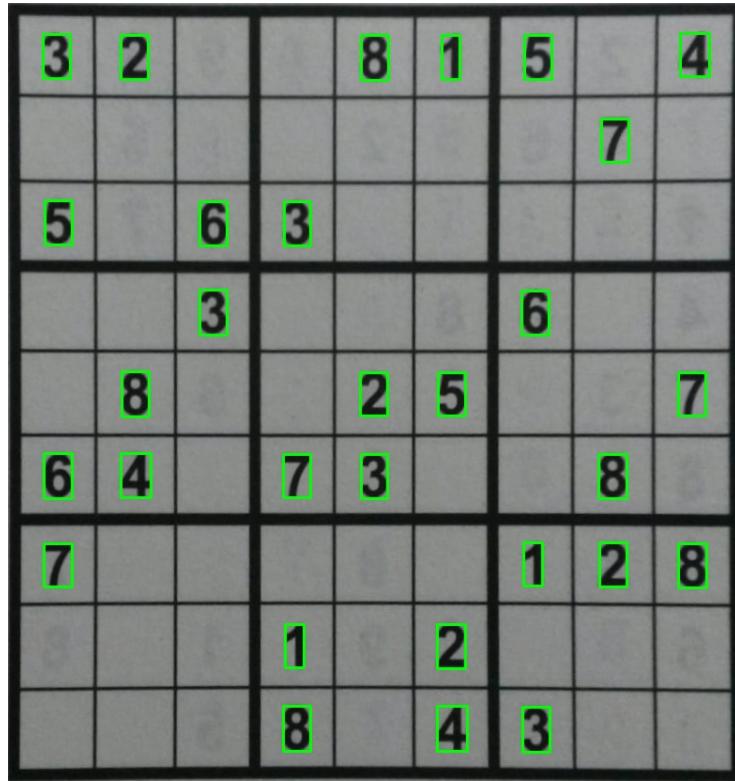


Figura 13: Rectángulos que contienen cada número del puzzle

El segundo paso es utilizar estos rectángulos para recortar el puzzle, obteniendo **regiones de interés** que contienen cada número. Realizaremos estos recortes sobre la imagen umbralizada, ya que nos interesa obtener una imagen binaria donde el número es una forma negra sobre fondo blanco:

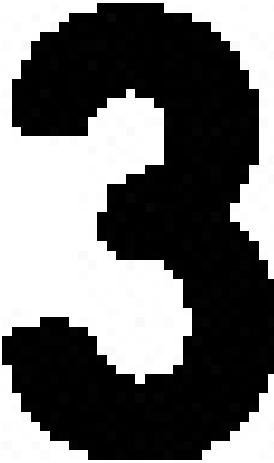


Figura 14: Ejemplo de región de interés extraída

```
x,y,w,h = cv2.boundingRect(contour)
number = cv2.bitwise_not(thresholded[y:y+h,x:x+w])
```

Código Fuente 13: Código que extrae la región de interés correspondiente al número y la convierte a negro sobre fondo blanco

El último paso es alimentar esta forma a un reconocedor óptico de números (OCR). Una opción es utilizar un reconocedor basado en máscaras, que reconoce el número según cómo de bien se aproxime la forma a una máscara predefinida de cada número. Mientras que esta opción es la más sencilla y rápida de implementar, es más complicado conseguir que generalice a distintas fuentes.

Una solución general es entrenar un clasificador mediante *Machine Learning* con un determinado conjunto de datos. Para elaborar y utilizar este modelo se ha utilizado la herramienta `tensorflow`, adaptando el modelo que trata de aprender el dataset `mnist` mediante redes convolucionales (<https://github.com/tensorflow/models/tree/master/official/mnist>).

2.3.2. Modelo

Inicialmente se ha utilizado dicho modelo entrenado con el dataset `mnist`, que contiene ejemplos de números manuscritos. El resultado de utilizar dicho modelo es bastante negativo, probablemente debido a que nuestro objetivo no es reconocer números manuscritos, sino impresos.

Por tanto se ha encontrado un dataset con números impresos, parte del dataset **Chars74K** [2] (<http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/>), correspondiente a las muestras de distintas fuentes de ordenador (FontEnglish), en cuatro estilos distintos, normal, negrita, cursiva, y negrita-cursiva.

Se ha adaptado el código de tensorflow para obtener los datos a partir de una clase propia, creada específicamente para leer las imágenes del dataset Chars74K y transformarlas a un formato introducible a la red neuronal. Este dataset se divide en datasets de entrenamiento y test de manera aleatoria con la proporción habitual de 66 % de instancias dedicadas a entrenar la red y 33 % a validarla.

Los pasos tomados para preparar las imágenes son:

1. Asignar la clase correspondiente a cada imagen, reconocible por el directorio en el que está.
2. Leer imagen en escala de grises a una matriz numpy mediante `cv2.imread(filename,0)`
3. Transformar la imagen a 28x28 píxeles mediante `cv2.resize(image, (28, 28), interpolation = cv2.INTER_LINEAR)`.
4. Convertir la matriz a tipo `float32` mediante `image.astype('float32')`.
5. Normalizar datos al rango [0.0,1.0] mediante `image/250.0`

Tras ello generamos un sampleado aleatorio de los índices de datos, para así conseguir datasets de entrenamiento y test:

```
idx = np.arange(0,len(images))
np.random.shuffle(idx)
split = int((2/3)*len(images))

train_idx = idx[0:split]
test_idx = idx[split:]
```

Código Fuente 14: Código utilizado para generar índices de entrenamiento y test

Estos índices se utilizan durante la llamada a las funciones `train` y `test`:

```

def train():
    x = images[train_idx]
    y = labels[train_idx]
    return tf.data.Dataset.from_tensor_slices(({'image':x},y))

def test():
    x = images[test_idx]
    y = labels[test_idx]

    return tf.data.Dataset.from_tensor_slices(({'image':x},y))

```

Código Fuente 15: Código que define las funciones que devuelven los datasets de entrenamiento y test

Estos datos se utilizan en el código adaptado originalmente para clasificado del dataset mnist. El modelo en su totalidad está en el fichero `cnn.py` del módulo `model`.

En resumen, el modelo se trata de una red neuronal formada por las siguientes capas:

1. Capa de entrada - output: 28x28.
2. Filtro convolucional - 32 filtros. Output: 28x28x32. Tamaño de kernel 5x5. Función de activación RELU.
3. Pooling - esta capa reduce el tamaño del input. Con un tamaño de 2x2 reduce a la mitad en la primera y segunda dimensión. Output: 14x14x32.
4. Filtro convolucional - 64 filtros. Output: 14x14x64. Tamaño de kernel 5x5. Función de Activación RELU.
5. Pooling. Output: 7x7x64.
6. Una capa que aplana el input (input 3D M x N x R se convierte en 1D M*N*R). Output: $7*7*64 = 3136$.
7. Capa oculta (ejecuta la operación $input \cdot w + bias$) - 1024 neuronas (tamaño output). Función de Activación RELU.
8. Dropout - técnica de regularización para reducir el sobreajuste.
9. Capa oculta - 10 neuronas (output). Función de Activación lineal.

En total el modelo entrena 3.274.634 parámetros. El output de esta última capa se alimenta a dos capas distintas, una calcula el *Argmax*, consiguiendo

así la clase a la que pertenece la instancia introducida. La otra capa calcula el *softmax*, que normaliza el vector de salida en una distribución de probabilidad. De esta manera se pueden también obtener las probabilidades que el modelo asigna a que la instancia introducida sea de cada una de las clases posibles.

Este modelo se ha entrenado con el dataset ya mencionado, con las instancias desde el 1 hasta el 9, tras un holdout aleatorio manteniendo el mismo número de instancias para cada clase.

Tras ello tenemos 677 instancias de cada clase para entrenamiento, y 339 para validación, con un total de 6.093 instancias de entrenamiento y 3.591 para validación. Se entrena el modelo durante 40 épocas, con un tamaño de batch de 100. Esto significa que utilizamos el dataset completo 40 veces, durante las cuales, cada 100 instancias, se calcula el gradiente del error y se actualizan los pesos.

Por tanto la red se somete a entrenamientos con 61 batches, para un total de 2.440 iteraciones de entrenamiento.

El valor final es de una precisión del 99,4% sobre el dataset de validación, y un valor para la entropía entre la distribución real y la predicha de 0,0249. Estos son, en general, valores muy positivos para un clasificador de este tipo.

Tras el entrenamiento, los pesos obtenidos se almacenan en un fichero con extensión .pb, que podemos leer dentro de una sesión de Tensorflow y obtener clasificaciones. El código completo para la lectura del modelo se encuentra en el fichero `numberclassifier.py` del módulo `models`. Este fichero contiene una función, `classifynumbers`, que devuelve la clase resultante de clasificar un array de imágenes tras su preprocessado.

Los pasos en dicho preprocessado son:

1. Modificar el tamaño a 20x20 píxeles.
2. Añadir un borde blanco tal que el tamaño final es de 28x28. Esto centra el número y lo hace más legible para la red.
3. Normalizar la imagen al rango [0,0,1,0].

```

h,w = img.shape[:2]
img = cv2.resize(img, (20, 20), interpolation = cv2.INTER_LINEAR)
img = cv2.copyMakeBorder(img, 4 , 4, 4, 4, cv2.BORDER_CONSTANT,
                        value=255)
img = img / 255

```

Código Fuente 16: Código que realiza el preprocessado de las imágenes para la red

Este preprocessado se aplica a cada región de interés, y estas se dan como input al modelo, que devuelve las clasificaciones.

2.4. Utilización del pipeline

El pipeline elaborado toma una imagen leída previamente mediante OpenCV e implementa dos funciones: `warpSudoku`, que devuelve el puzzle recortado, y `extractNumbers`, que toma dicho recorte y devuelve una matriz conteniendo el sudoku en el orden encontrado.

El paquete `autosudoku` contiene una clase `__main__.py` que demuestra un posible uso de dicha pipeline. El programa toma como argumento el path a una imagen y tras leerla aplica el pipeline para obtener el sudoku. Imprime la matriz por consola y muestra la imagen original con el sudoku impreso en su centro, lo que permite comprobar si es correcto más fácilmente:

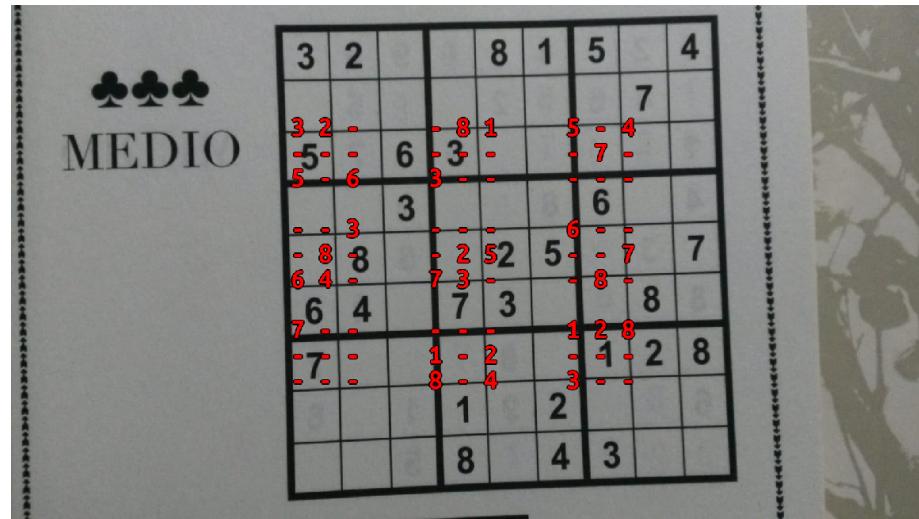


Figura 15: Sudoku obtenido impreso sobre la imagen original.

3	2	-	-	8	1	-	5	-	4
-	-	-	-	-	-	-	-	7	-
5	-	6	-	3	-	-	-	-	-
-	-	3	-	-	-	-	6	-	-
-	8	-	-	2	5	-	-	7	-
6	4	-	-	7	3	-	-	8	-
7	-	-	-	-	-	-	1	2	8
-	-	-	-	1	-	2	-	-	-
-	-	-	-	8	-	4	-	3	-

Figura 16: Sudoku obtenido impreso en consola.

La inclusión de texto en la imagen se realiza mediante el paquete `Pillow`, aunque OpenCV también permite dibujar texto sobre una imagen, ya que `Pillow` permite utilizar fuentes importadas, y era necesaria una fuente que mostrara correctamente caractéres especiales (tabuladores y saltos de línea, concretamente), que la fuente de OpenCV no era capaz de gestionar.

3. Utilización del programa

En esta sección se describe brevemente cómo instalar las dependencias necesarias y cómo utilizar el programa elaborado.

Primeramente es recomendable instalar el gestor de paquetes de python conda <https://conda.io/>. Tras ello se puede ejecutar en una terminal la orden `conda env create -f environment.yml` desde la carpeta raíz de esta entrega. **Esto es importante**, ya que este programa utiliza una versión de OpenCV específica, la obtenida mediante la orden `pip 'opencv-python<4'`. Cualquier ejecución del programa con un intérprete distinto al instalado por conda mediante el fichero `.yml` puede no funcionar correctamente.

Esta orden crea un nuevo entorno de nombre `autosudoku` con todas las dependencias necesarias. Para activar el entorno utilizar la orden indicada por conda tras la instalación (p.e. `conda activate autosudoku`).

Una vez activado el entorno, desde la carpeta raíz del proyecto se puede utilizar la orden `python autosudoku IMAGEN`, con `IMAGEN` siendo el path a una imagen (p.e. `sample-images/02.jpeg`).

Esto ejecutará el fichero `__main__.py` del paquete `autosudoku`, que tiene la funcionalidad indicada anteriormente.

También es posible escribir un script que utilice la clase `Pipe` directamente.

Se ha comprobado el correcto funcionamiento del sistema en varias distribuciones Linux (basadas en Ubuntu y Arch) y Windows 10.

Para ejecutar el script `cnn.py` del módulo `model` (red neuronal que genera el modelo utilizado para clasificar los números) es necesario clonar el repositorio <https://github.com/tensorflow/models/> y añadir el directorio en el que se sitúa esta carpeta al `PYTHONPATH`, así como descargar el dataset EnglishFont, parte de Chars74K [2]. Sin embargo el modelo ha sido pre-entrenado y guardado en otra carpeta accesible al programa, por lo que esto no es necesario.

4. Resultados y conclusiones

Durante la realización del trabajo se ha tratado de hacer la solución lo más general posible, tal que funcionara en el mayor número de conjuntos de imágenes posible.

Aunque la pipeline funciona bien en muchas de las imágenes de las que se dispone, en las últimas etapas de la pipeline sólo se consiguen resultados correctos con las imágenes de un tipo, caracterizadas por tener bordes bastante más gruesos. Además, el programa tiene dificultades con imágenes cuya perspectiva no es completamente plana o en las que el borde exterior del sudoku aparece cortado. En general los errores se deben a la detección no completa de los contornos. Sin embargo, la utilización de una red neuronal generaliza la extracción de números.

Un posible trabajo futuro asociado es tratar de aplicar otras técnicas de detección de contornos, como son umbralizados adaptativos alternativos al método Otsu, como se ha tratado de hacer para el recorte del puzzle, o detección de bordes mediante Canny.

Referencias

- [1] Adrian Rosebrock. OpenCV shape detection, 2016. <https://www.pyimagesearch.com/2016/02/08/opencv-shape-detection/> [Visitado última vez 16 de enero de 2019].

- [2] T. E. de Campos, B. R. Babu, and M. Varma. Character recognition in natural images. In *Proceedings of the International Conference on Computer Vision Theory and Applications, Lisbon, Portugal*, February 2009.