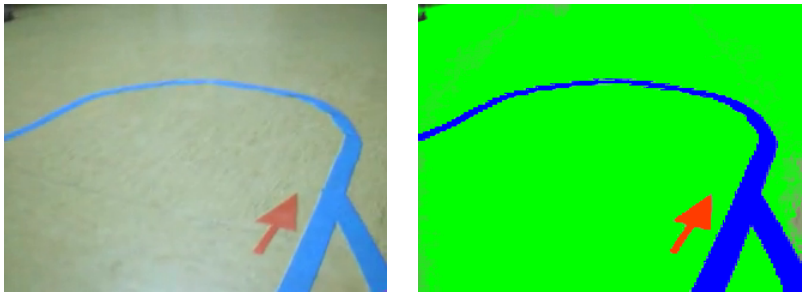
Análisis de imágenes
Robótica y Percepción Computacional

Daniel Fernández
Miguel García-Mauriño

8 de abril de 2019

1. Algoritmo de segmentación de imágenes

Para la segmentación de las imágenes se utiliza un clasificador por color basado en la distancia euclídea. Para entrenarlo, se han tomado varias capturas del vídeo y se han coloreado de forma que los píxeles de distinto color correspondan a objetos de distintas categorías.



Cuadro 1: Ejemplo de una imagen original y otra coloreada, como las que se han usado para entrenar al clasificador

Comparando las capturas originales con las coloreadas, se crea un dataset de los colores en coordenadas rgb normalizadas que tienen los objetos de cada categoría en la imagen. Finalmente, se halla el centroide de cada categoría calculando la media de los colores de todos sus puntos.

Cuando llega el momento de la clasificación, se recuperan los valores de los centroides previamente calculados y se calcula de forma vectorizada la distancia de cada uno de los puntos de la imagen a cada uno de los centroides para después tomar aquel con el que tenga menor distancia.

El código está dividido en prac_ent.py, que realiza el entrenamiento, rellenando los parámetros en parametros.py, que luego lee clasificador.py al ejecutarse. El script prac_run.py utiliza el clasificador definido en clasificador.py para hacer la segmentacio y el análisis de imagen

El clasificador ha tardado de media unos 0.007 segundos en cien ejecuciones sobre diez imágenes distintas.

1.1. Código Python

El clasificador se plantea siguiendo la función **??**.

$$Z_i^T \cdot X - 1/2 \cdot Z_i^T \cdot Z_i \tag{1}$$

El clasificador obtiene 3 centroides desde el archivo parametros.py
Luego se calcula el vector W = (ZiT * Zi * -1/2,z1,z2,....).

```
self.c_marca = parametros.c_marca
self.c_fondo = parametros.c_fondo
self.c_linea = parametros.c_linea
self.centroids = np.array([self.c_marca, self.c_fondo, self.c_linea])
self.W = np.c_[np.apply_along_axis(lambda zi: zi.dot(zi)/-2,1,self.centroi
```

Para clasificar se normaliza la imagen recibida, se hace un cambio de forma de la matriz se calcula la distancia a los vectores añadiendo al vector X un 1 al principio X = (1,x1,x2,...). Se hace la multiplicación matricial de W y X. Y finalmente se selecciona la clase con el valor mas alto.

```
imn = np.rollaxis((np.rollaxis(im,2)+0.0)/np.sum(im,2),0,3)[:,:,:2]
x,y,z = imn.shape
imr = np.reshape (imn,(-1,z))
X = np.insert(x,0,1,axis=1)
X = X.dot(self.W.T)
cats = np.argmax(d,axis=1)
res = np.reshape (cats,(x,y))
```

Archivo prac_ent.py se encarga de leer las imágenes que hay en la carpeta y comprobar si ya se han calculado previamente los centroides, en caso de que ya estén calculados se cargan los datos y se calculan los centroides.

Nomraliza la imagen original, despues asigna a cada pixel su clase. Se guarda en un fichero los valores y la clase a la que pertenece.

```
imn = np.rollaxis((np.rollaxis(im,2)+0.0)/np.sum(im,2),0,3)[:,:,:2]

data_marca+=imn[np.where(np.all(np.equal(mark,(255,60,0)),2))].tolist()
data_fondo+=imn[np.where(np.all(np.equal(mark,(0,255,0)),2))].tolist()
data_linea+=imn[np.where(np.all(np.equal(mark,(0,0,255)),2))].tolist()

with open("dataset.txt","w") as f:
        for r,g in data_marca:
                f.write("{},{},0\n".format(r,g))
        for r,g in data_fondo:
                f.write("{},{},1\n".format(r,g))
        for r,g in data_linea:
                f.write("{},{},2\n".format(r,g))
```

Finalmente se calcula la media de cada clase.

```
c_marca = np.mean(data_marca,0)
c_fondo = np.mean(data_fondo,0)
c_linea = np.mean(data_linea,0)
```

1.2. Tiempos de ejecución

Video1 tiene un tiempo de ejecución de 24.56s
Line0 tiene un tiempo de ejecución de 29.44s
Video2017-3 tiene un tiempo de ejecución de 37.58s
Video2017-4 tiene un tiempo de ejecución de 30.01s
Tiempo medio en procesar una imagen 0.10s.

2. Algoritmo para medir distancias

Para medir la distancia de un objeto de tamaño conocido a la cámara, en este caso una pelota, primero tomamos una captura del objeto a una distancia fija. Con estos datos podemos calcular la distancia focal:

$$f = \frac{w \times d}{W} \quad f = \frac{h \times d}{H} \tag{2}$$

Donde d es la distancia a la que se hallaba la pelota en el momento de la captura, w y h son, respectivamente, la anchura y altura aparentes de la pelota en la imagen, mientras que W y H son la anchura y altura reales. Evidentemente, la distancia focal se podría calcular simplemente con un par w,W o h,H, pero se utilizan las dos por redundancia y se calcula la media para resolver el problema de ajuste.

Una vez conocida la distancia focal de la cámara y sabiendo que el tamaño de la pelota no cambia en el mundo real, basta con despejar la distancia de (1) para saber la distancia de la pelota a la cámara en un instante determinado.

$$d = \frac{W \times f}{w} \quad d = \frac{H \times f}{h} \tag{3}$$

Para la ejecución de este código se utiliza el clasificador que se ha explicado en el apartado anterior para diferenciar la pelota de aquello que no lo es. Después, se encuentra un rectángulo que la rodea, y se utiliza la altura y anchura de dicho rectángulo para calcular la distancia como se ha explicado en la ecuacion (2).

2.1. Código Python

Para obetner las esquinas del rectángulo que inscribe la pelota y su centro, se buscan en la imagen las coordenadas de la categoría de la pelota.

Se calcula el centro con la media de las coordenadas y se obtienen las esquinas.

```
# Encontrar las coordenadas de la imagen que pertenezcan a la categorí
dr = (0,0)
ul = (0,0)
center = (0,0)

ball = np.array(np.where(im==1))
if not ball.any():
    return ul,dr,center
# El centro es la media de las coordenadas
center = np.uint(np.mean(ball,1)).tolist()
# la esquina superior izquierda son las coordenadas de menor valor, la
# son las coordenadas mayores
dr = np.max(ball,1).tolist()
ul = np.min(ball,1).tolist()

center.reverse()
ul.reverse()
dr.reverse()
return tuple(ul), tuple(dr), tuple(center)
```

Para calcular la distancia focal de la cámara se calcula la anchura y altura aparente de la pelota en la imagen. Despues calcular las dos medidas de la focal disponibles y por último se calcula la media.

```
# obtiene las esquinas del rectángulo que modela la pelota
ul,dr,_ = ball_square(im)

# calcula la altura y anchura aparente de la pelota en la imagen
apparent_width = abs(ul[0]-dr[0])
apparent_heigth = abs(ul[1]-dr[1])

# calcula dos posibilidades de la focal, ya que conocemos dos medidas
focal_width = float(apparent_width * known_distance) / KNOWN_WIDTH
focal_heigth = float(apparent_heigth * known_distance) / KNOWN_HEIGTH
```

```
    # devuelve la media por redundancia
    return np.mean([focal_heigth, focal_width])
```

3. Algoritmo de análisis de imagen

Para realizar el análisis de la imagen se ha partido de una serie de asunciones:

1. Las líneas son infinitas
2. No se pueden ver segmentos pasados ni futuros
3. El horizonte queda por encima del campo de visión de la cámara, es decir, todo lo que ve la cámara está en el suelo.
4. La línea sobre la que está el robot cuando empieza a moverse se encuentra centrada en la imagen
5. La línea sobre la que está el robot cuando empieza a moverse está en la parte baja de la imagen

Las tres primeras parecen razonables, mientras que las dos últimas, pese a que podrían no serlo, son necesarias (ver seccion 3.3)

3.1. Tipo de línea

Tenemos tres escenarios posibles para la línea:

- Una sola lína
- Una bifurcación o cruce en T
- Un cruce en X

Para diferenciar entre cada uno de ellos contamos el número de contornos distinos que no son una línea. Así, si hay tres o más podemos decir que estamos en una bifurcación y en una X, respectivamente, pero si se encuentran más contornos probablemente signifique que ha habido un error.

Sin embargo, Si menos de tres contornos, significa que estamos ante una sola línea. Para comprobar si es recta o curva encontramos el cierre convexo de dicha línea y calculamos su área. Si es parecida al área de la línea entonces consideramos que nos hallamos ante una linea recta. Si es distinta, consideramos que es una curva.

3.2. Orientación de la flecha

Para encontrar la orientación de la flecha dividimos el problema en dos subproblemas. El primero es encontrar la dirección y el segundo el sentido.

3.2.1. Dirección

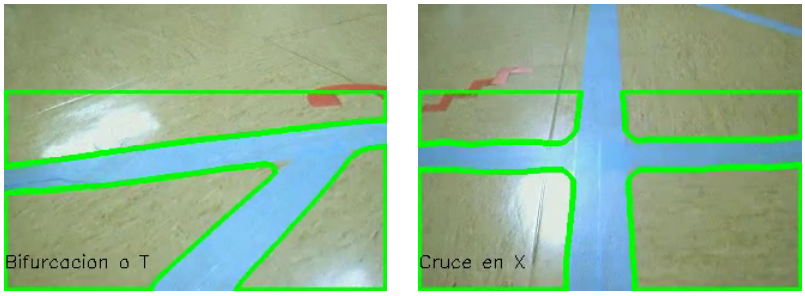
La dirección de la flecha viene dada por la función de opencv fitLine, que devuelve los parámetros de una recta que pasa por el centro de la figura y está orientada de forma que se minimiza su distancia con todos los puntos de la flecha.

3.2.2. Sentido

Concidos los parámetros de la recta y el punto central de la flecha, para hallar el sentido de la misma se parte de la suposición de que habrá más área en el contorno de la "cabeza" de la flecha que en el de la "cola". Así, calculamos la perpendicular a la recta de la dirección y calculamos el área del contorno queda a cada lado de la perpendicular. Consideramos que la flecha apunta en el sentido de la mitad de mayor área.

3.3. Línea de entrada y de salida

Podemos adaptar las suposiciones explicadas anteriormente para el problema de la línea de entrada y de salida. En nuestro caso, suponemos que la entrada de la primera ejecución, es decir, del primer frame de vídeo, se encuentra en el punto de línea más cercano al centro del borde inferior de la imágen. Para encontrar la salida diferenciamos dos situaciones:



Cuadro 2: Detección de bifurcaciones y cruces

- **Una línea**: Si hay una sola línea, suponemos que sólo hay dos salidas posibles: la entrada actual y la salida por la que hay que salir. Así, el punto de línea más lejano al de entrada que se encuentre en un borde será un punto de la salida por la que debe progresar el robot.
- **Varias líneas**: Si hay varias línas, suponemos que hay alguna flecha, por lo que calculamos su orientación y prolongamos la recta resultante en la dirección pertinente hasta que corte con uno de los bordes de la imágen. El punto de línea más cercano al punto de corte se considera la salida por la que debe seguir el robot.

3.4. Código Python

Para conocer que tipo de línea se esta visualizando, si una recta, una curva, un cruce. Primero se cuentan los contornos que no pertenezcan a la línea. Si hay 4 o 3 contornos es una bifurcación.

```
conts, hier = cv2.findContours((img == 0).astype(np.uint8)*255,cv2.RETR_LI
```

Para discriminar las rectas de las líneas se comprueba si el contorno y el chull tienen un área aproximada.

```
chull = cv2.convexHull(line)
charea = chull_area(chull)
if area > charea * (1-thres) and area < charea*(1+thres):
    return LINEA_RECTA
```

Por último si es una curva hay que saber la dirección, para ello se comprueba la cantidad de píxeles de línea en las dos mitades de la imagen.

```
medio = int(img.shape[1]/2)
derecha = img[:,medio:]
izquierda = img[:,:medio]
if np.size(derecha[derecha == 1]) > np.size(izquierda[izquierda == 1]):
    # hay mas linea a la derecha que a la izquierda -> gira a la derecha
    return CURVA_DERECHA
return CURVA_IZQUIERDA
```

Para saber la entrada y la salida se cogen los bordes de la imagen, se miden las distancias con la entrada anterior, la más cercana es la entrada.

```
bordes = in_border (linea).T
distancias = np.sum((bordes - anterior_entrada)**2, axis=1)
if np.size (distancias)> 0:
```

```
    cercano = np.argmin (distancias)
    entrada = bordes [cercano]
else:
    entrada = (0,0)
```

Si hay más de una salida es una bifurcación y hay una flecha, por lo que se comprueba la dirección de la flecha, se calcula por dónde sale de la imagen, y se escoge el punto de línea más cercano al punto de salida.