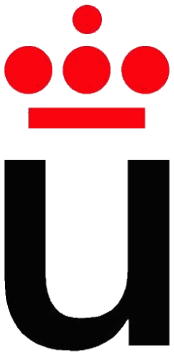


Universidad Rey Juan Carlos

Asignatura: Visión Artificial

PRÁCTICA 4

Segmentación y reconocimiento de caracteres

Gabriel Chica Fernández – GIC + GII – Expediente 53
Rafael J. Vicente García – GIC + GII – Expediente 76

22/05/2016

ÍNDICE

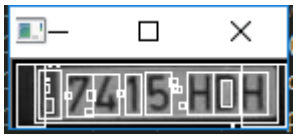
Práctica IV: Segmentación y reconocimiento de caracteres

1. Localización de los dígitos de la matrícula (car.py)

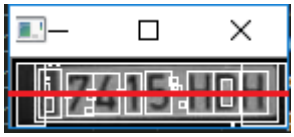
Se localizarán los coches existentes en la imagen siguiendo los pasos realizados en el desarrollo de la práctica III y haciendo uso del clasificador CascadeClassifier definido en el fichero *resources/coches.xml*.

Una vez se han encontrado los coches, se procede a localizar las matrículas en la imagen. Para ello se utilizará el clasificador CascadeClassifier definido en el fichero *resources/matriculas.xml*. Una vez se han extraído las matrículas del coche, se procede de la siguiente manera para cada una:

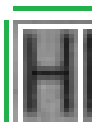
1. Se umbraliza la imagen de la placa con *cv2.adaptativeThreshold* y se localizan los contornos de la imagen con *cv2.findContours:*
2. Para cada contorno, se obtiene el rectángulo que lo contiene con *cv2.boundingRect* y se almacena en una lista.



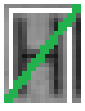
3. Se filtran los contornos de la lista en tres pasos:
    a. Descartamos todos los rectángulos que no sean cruzados por una línea a media altura de la placa o que sean demasiado pequeños (el límite de altura *H_MIN* es un tercio de la altura de la placa).



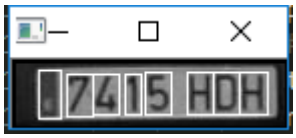
    b. Ordenamos los rectángulos por ratio (longitud / altura) y dividimos en grupos, creando un nuevo grupo cada vez que se produzca una diferencia superior a *ratioLimit* respecto al nuevo rectángulo. Seleccionamos el grupo con más elementos.



    c. Ordenamos los elementos restantes por hipotenusa, repitiendo el paso anterior pero buscando contrastes por hipotenusa (*hypotenuseLimit)*, para descartar posibles elementos de igual ratio pero de distintas dimensiones.



    d. Se devuelve la lista de rectángulos resultante:

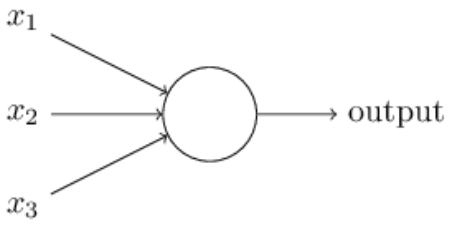


2. Entrenamiento de la Red Neuronal para procesar caracteres (neural.py)

Para realizar el procesado de caracteres se ha decidido experimentar con una red neuronal. Para ello, hemos aprendido todo lo necesario del libro online gratuito neuralnetworksanddeeplearning.com, y hemos usado la misma librería con la que en él se trabaja adaptándola a nuestras necesidades: *neural-networks-and-deep-learning*. Más información en la bibliografía.

2.1 Conceptos básicos

Nuestra red neuronal está formada por sigmoid neurons, este es el modelo más utilizado a día de hoy y nos permite que al realizar pequeñas modificaciones en los parámetros neuronales (weights and biases) se produzca un cambio igualmente pequeño en la salida, lo cual permite adaptar el sistema de forma rápida en función de los resultados.



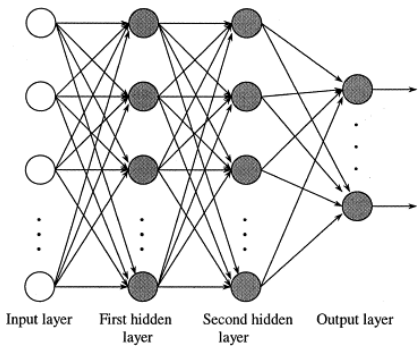
x1, x2, x3 son los valores de entrada de la neurona, y pueden tomar valores decimales entre 0 y 1. La salida viene determinada por el valor de retorno de la función sigmoid:

$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

Donde wj son los pesos asignados a cada entrada, xj el valor de cada entrada y b (bias) el valor delimitador de la neurona.

Las neuronas se organizan en capas. Siempre tendremos una capa de entrada, que en el caso de procesar imágenes debería ser igual al número de píxeles de entrada, y una de salida, que debería tener tantas neuronas como posibles resultados. Las capas internas se denominan capas ocultas y añaden complejidad al sistema con la que mejorar la precisión del mismo.

En nuestro caso, hemos utilizado una red con 100 neuronas de entrada, 100 neuronas de primer nivel, 100 neuronas de segundo nivel y 37 neuronas de salida. Esta ha sido la configuración que mejores resultados nos ha dado, con una tasa de éxito de un 90% en las imágenes de test (menor en algunos rectángulos de caracteres de menor resolución).



2.1 Adaptando el conjunto de imágenes de entrenamiento a la red neuronal

La red neuronal debe recibir una lista de tuplas (x, y) de entrenamiento, donde 'x' es una imagen de entrenamiento umbralizada y redimensionada a 100x1 (esto es, un vector de 100 dimensiones), e 'y' es un vector 37x1 con un valor de 1.0 en el índice que representa el valor de la imagen (por ejemplo, el carácter '0' asociado al número 0 se representa como un 1.0 en la posición 0 y el símbolo 'ESP' se representa como un 1.0 en la posición 36).

Adicionalmente, puede recibir una lista de tuplas de testing o validación (que en este caso son los rectángulos de caracteres extraídos de las placas previamente y guardados con *cv2.imwrite*), que no entrena a la red neuronal, pero sirve para monitorizar el progreso real. En este caso la componente 'y' no es un vector, sino el índice que debería ocupar.

Nuestra red neuronal tiene que distinguir entre 37 posibles valores (0-9 + A-Z + 'ESP') contando con 250 ejemplos de cada uno, con una resolución muy pequeña (100 píxeles). Este conjunto de ejemplos es demasiado reducido para la red, que en pruebas apenas lograba una tasa de éxito del 50% sobre datos de test. Se barajaron distintas opciones y finalmente se optó por ampliarlo con la técnica de crear cuatro nuevas imágenes a partir de cada una, desplazando la original un píxel hacia arriba, abajo, izquierda y derecha.

```
for d, axis, index_position, index in [
        (1,  0, "first", 0),
        (-1, 0, "first", SIDE_SIZE - 1),
        (1,  1, "last",  0),
        (-1, 1, "last",  SIDE_SIZE - 1)]:
    new_img = np.roll(image, d, axis)
    p = np.empty(SIDE_SIZE)
    p.fill(C)
    if index_position == "first":
        p = np.empty(SIDE_SIZE)
        p.fill(C)
        new_img[index, :] = p
    else:
        new_img[:, index] = p
```

De esta forma, logramos tener para cada carácter las originales 250 imágenes más las 1000 generadas. En total, pasamos de un conjunto de 9250 imágenes a uno de 46250 que nos permite alcanzar la tasa de éxito del 90% sobre datos de test.

Épocas de entrenamiento 17-19 de un total de 100 (en las últimas fases se llegó a 165 / 180 en evaluation data y a 43000 / 46250 en training data). Las imágenes de evaluación son caracteres recortados de las matrículas de testing_ocr.

```
Epoch 17 training complete
Accuracy on training data: 39393 / 46250
Accuracy on evaluation data: 113 / 180

Epoch 18 training complete
Accuracy on training data: 39900 / 46250
Accuracy on evaluation data: 117 / 180

Epoch 19 training complete
Accuracy on training data: 39766 / 46250
Accuracy on evaluation data: 134 / 180
```

Esta red neuronal está guardada en el fichero *resources/neural_net* y es la que utilizaremos para procesar los caracteres extraídos de la placa del coche.

3. Identificación de los caracteres de la placa del coche

Una vez entrenada la red neuronal sólo queda utilizarla para procesar cada rectángulo-caracter extraído de la placa del coche. Esto lo logramos una vez cargada la red desde fichero y filtrados los rectángulos sobrantes de la siguiente forma:

```
for r in max_rs:
    crop_img = max_plateRectangle[r.y:r.y+r.h, r.x:r.x+r.w]
    p_img = processImageForNeuralNet(crop_img, image=True)
    print "{}".format(translateNeuralOutput(np.argmax(net.feedforward(p_img)))) #Neural Network Output
```

Para cada rectángulo *r* extraemos sus pixeles en la imagen *crop_img* y la convertimos al formato aceptado por la red neuronal llamando a *processImageForNeuralNet* (que devuelve un vector de 100 x 1). Finalmente, se pasa el vector de píxeles a la red neuronal, que devuelve un vector con valores de salida de entre los cuales se obtiene el índice con el valor más alto. Tomando la tabla ASCII como referencia podemos traducir el valor entero al carácter deseado:

```
def translateNeuralOutput(value):
    if(value < 10):
        return chr(value + 48)
    if(value == 36):
        return 'ESP'
    return chr(value + 55)
```

El resultado es el siguiente: