Universidad Francisco Marroquín

Computer Vision

Carlos Cujcuj

José Guzmán

Laboratorio 2

Reconocimiento de Logo Universidad Francisco Marroquín

Con el objetivo de localizar el logo de la UFM dentro del alcance de una cámara de video, se aplicaron distintas herramientas de procesamiento de imágenes, entre estas se encuentran:

- Conversión de Colores
- Edge Detection con función Canny
- Pirámides de Imagen
- Template Matching
- Optimizaciones de Cython
- Optimizaciones de Multithreading

Como parte del proceso de la detección del logo, lo primero que se hace es una serie de modificaciones a la imagen del logo que se usará como modelo para la función de Template Matching. Con el propósito de aumentar la precisión de la detección y reducir cualquier detalle ruidoso, se realiza la lectura del logo con el parámetro *cv.IMREAD_GRAYSCALE* y posteriormente se le aplica la función *Canny()*. Esta función hace un conjunto de transformaciones sobre el logo en blanco y negro. El primer parámetro de la función es el arreglo que representa la imagen, los otros dos parámetros son valores de *thresholding* que representan el valor mínimo y máximo dentro del cual debe de entrar un resultado del gradiente para ser considerado una orilla. Se estableció un área de aceptación entre 50 y 200 para asegurarse de capturar la mayor cantidad de detalle (en cuestión de orillas) al analizar los frames del video.

Una vez el logo esté en su estado final para la comparación, se guarda un conjunto de imágenes que representan la pirámide del mismo. Esto se realiza con la función *cv.resize()* en la que se ingresa un factor por el que se va a multiplicar cada eje de la imagen y el tipo de interpolación a utilizar. La función externa de pyramid() acepta un parámetro de imagen, otro de escala (factor) y otro de tamaño mínimo de las imágenes. En este caso se utilizó un factor de 0.95 con la intención de detectar el logo prácticamente en cualquier tamaño hasta llegar al tamaño mínimo por defecto de (32,32). Al momento de entrar al ciclo de ejecución de la cámara de video, se realizan las mismas operaciones que se le hicieron al logo a cada frame, con la excepción de la pirámide. Posteriormente, el programa entra en un ciclo en el que aplica la función *cv.matchTemplate().* Esta función se encarga de recorrer la imagen en busca del modelo que se provee. Los parámetros que acepta incluyen la imagen, el modelo y el tipo de operación que se realiza para la verificación. En este caso se decidió utilizar el método de *cv.TM_CCOEFF_NORMED*, esta decisión se tomó en base al rendimiento del método y al tipo de dato que devuelve al momento de aplicar *cv.minMaxLoc()* sobre la respuesta.

Debido a que en esta implementación se está haciendo la operación de *cv.matchTemplate()* con diferentes modelos, es necesario guardar cada posible valor máximo de la operación. Si se utiliza cualquiera de los métodos que no está normalizado, el valor máximo depende del tamaño del modelo. Por lo tanto, se decidió utilizar la versión normalizada de COEFF para comparar los datos resultantes de manera directa. Al momento de salir del ciclo interno del análisis de todas las diferentes modelos, el programa obtiene la localidad que tiene la mayor probabilidad de tener el objeto deseado. Sin embargo, se aplica un threshold con el valor de 0.4. Esto significa que cualquier valor máximo por arriba de 0.4 es lo suficientemente alto para aceptarlo y clasificarlo como el logo que se está buscando.

El detector básico tal como se explica anteriormente tiene un rendimiento de aproximadamente **0.90 frames por segundo.** Este valor es muy bajo, esto se debe a la cantidad de veces que se aplica una operación sobre todos los pixeles de cada frame.

Uno de los métodos de optimización utilizados es Cython. Cython funciona cambiando código de Python a código de C cuando se escribe. Esto es debido a que Python es primero interpretado y luego se convierte a código de C para ser ejecutado, en otras palabras, se evitan abstracciones del lenguaje de Python para ejecutar directamente en C. Para el proyecto utilizamos Cython en las declaraciones de variables int, double, list y tuplas como (int, int) con una declaración de forma cdef el cual pareciera que es Python pero en realidad se está escribiendo en C. También para optimizar las funciones se declararon el tipo de variable a utilizar en los parámetros que se le asignan. Y en algunos parámetros de array se utilizó código de Numpy por parte de Cython de forma cnp.ndarray importando con cimport. Por parte de la validación de indices, se optimizo con decoradores declarados como @cython.boundscheck(False) los cuales nos ayudaron a desactivar las validaciones existe un out of bounds y @cython.wraparound(False) para la validación en no utilizar índices negativos. Para algunas funciones no fue posible utilizar cython debido al tipo de dato que devuelve en su return como en el caso de yield el cual es un operador nativo de Python y también en funciones que devolvían tipo de dato de OpenCV como en el caso de cv.VideoCapture(0) que nos devolvía un tipo de dato <class cv.VideoCapture>. Esto afecto de gran manera la optimización debido a que muchas arrays provenían de esta librería a las cuales no es posible acceder y declarar el tipo de variables u otros parámetros debido a que provienen nativamente de OpenCV.

El siguiente método de optimización que se utilizó fue el de threading. El ciclo interno de evaluación del modelo con la función de Template Matching se puede separar en distintos flujos de ejecución. Esto se debe a que cada llamada a la función de Template Matching con un tamaño diferente de logo es independiente de las demás. Se realizaron pruebas con la cantidad de threads necesarios para aumentar el rendimiento del programa. Un valor bajo no representa un cambio en el rendimiento mientras que un valor alto gasta recursos en el manejo de threads y hay un retorno decreciente. El valor que se utilizó fue de 7 threads, por lo tanto, si la pirámide genera 28 imágenes, cada thread aplica la función de Template Matching sobre 4 prácticamente al mismo tiempo. Esta optimización causó un aumento de aproximadamente **3 frames por segundo** en la ejecución. El aumento en el rendimiento no es masivo, sin embargo, la diferencia sí es clara, pues el detector básico no pasa del frame por segundo.