Análisis de microcontroladores
Informe final

Juan Ramírez , Natalia López
Bogotá , Universidad distrital Francisco José de caldas, Colombia
nalopezb@udistrital.edu.co
jueramirezs@udistrital.edu.co

Resumen- Este artículo presenta el diseño y desarrollo de un carrito seguidor de línea que utiliza la cámara OV7670 y Raspberry Pi Pico W. El sistema sigue la línea utilizando técnicas de procesamiento de imágenes y redes neuronales convolucionales (CNN). Se describen los componentes del hardware, la arquitectura de la red neuronal, y los algoritmos de control implementados en micro python. Los resultados experimentales demuestran la eficacia del enfoque basado en redes neuronales en comparación con métodos tradicionales.

Palabras clave: Carrito seguidor de línea, Raspberry Pi Pico W, cámara OV7670, redes neuronales, procesamiento de imágenes, CNN.

I. INTRODUCCIÓN

En el campo de la robótica móvil, los carritos seguidores de línea son una de las aplicaciones más básicas y fundamentales. Estos robots están diseñados para seguir una trayectoria predefinida marcada por una línea en el suelo, generalmente negra sobre un fondo blanco, utilizando diversos métodos de detección y control. Tradicionalmente, estos sistemas han dependido de sensores infrarrojos para detectar la línea. Sin embargo, estos métodos pueden ser limitados en términos de precisión y adaptabilidad a entornos variados.

El avance de la tecnología de procesamiento de imágenes y aprendizaje automático ha permitido la implementación de métodos más avanzados y robustos para la detección y seguimiento de líneas. Este artículo presenta el diseño y desarrollo de un carrito seguidor de línea que utiliza una cámara OV7670 y una Raspberry Pi Pico W, aprovechando técnicas de redes neuronales convolucionales (CNN) para mejorar la precisión y adaptabilidad del seguimiento.

La elección de una cámara para la detección de la línea en lugar de sensores infrarrojos permite una mayor flexibilidad y capacidad de adaptación a diferentes tipos de trayectorias y condiciones de iluminación. Además, la implementación de redes neuronales convolucionales proporciona una mejora significativa en la capacidad de procesamiento de imágenes, permitiendo al sistema aprender y adaptarse a diversas condiciones del entorno.

II. METODOLOGÍA

Daremos una breve **descripción del sistema implementado** el cual consta en primer lugar de un chasis de cuatro ruedas que contiene dos motores DC, dos Raspberry Pi Pico W una para la toma de decisiones en cuanto al control de los motores y otra para el control de la cámara OV7670 y procesamiento de imágenes, un módulo controlador de motores L298N el cual junto con la Raspberry manejan el movimiento del carrito y sus decisiones.

La siguiente parte fundamental del proyecto es la **adquisición de los datos** procedentes de la imagen que captura la cámara, la cual está ubicada en la parte frontal del carro mirando directamente a la línea ubicada dentro de la pista, para entender cómo obtuvimos los datos de la cámara hay que hablar un poco más teóricamente de cómo funciona.

Cámara OV7670

Fig 1. Cámara OV7670

La cámara OV7670 es un sensor de imagen CMOS que captura imágenes en formato VGA y puede ser configurada para diversas resoluciones y formatos de salida, la cámara se utiliza para capturar imágenes en tiempo real del entorno.

La cámara captura las imágenes por medio de un sensor el cual convierte la luz entrante en señales eléctricas que luego se procesan para formar una imagen digital, la resolución es de 640x480. Para simplificar el procesamiento de imágenes, las imágenes capturadas se convierten a escala de grises. Esto reduce la cantidad de datos y facilita la detección de la línea, que generalmente tiene un contraste alto con el fondo.

Mediante el lenguaje de programación circuit python que contiene la librería adafruit con la cual podemos controlar la cámara y en el entorno de Thony realizamos el código que se presenta a continuación:

En este código se observa que en las primeras líneas se importan los módulos necesarios “sys” maneja el sistema y argumentos de línea de comandos, time gestiona funciones de tiempo, “digitalio” maneja E/S digitales, “busio” maneja buses de comunicación I2C/UART/SPI y board define los pines específicos de la placa.

```
import sys
import time
```

```
import digitalio
import busio
import board
from adafruit_ov7670 import (
    OV7670,
    OV7670_SIZE_DIV16,
    OV7670_COLOR_YUV,)
```

Luego de ello por medio del bus de comunicación I2C se hace la configuración de comunicación entre la Raspberry Pi Pico W y la cámara

```
# Configuración de la cámara
cam_bus = busio.I2C(board.GP21, board.GP20)
```

Posterior a ello se inicializa la cámara con todos los parámetros de configuración

```
cam = OV7670(
    cam_bus, #bus I2C configurado
    data_pins=[ #Pines de datos
        board.GP0,
        board.GP1,
        board.GP2,
        board.GP3,
        board.GP4,
        board.GP5,
        board.GP6,
        board.GP7,
    ],
    clock=board.GP8, #Reloj
    vsync=board.GP13, #Sincronización vertical
    href=board.GP12, #Referencia horizontal
    mclk=board.GP9, #Reloj-maestro
    shutdown=board.GP15, #Apagado
    reset=board.GP14, #Reinicio
)
```

Se configura el tamaño de la imagen y se invierte verticalmente, se configura el espacio de color de la cámara a YUV que descompone los colores en tres componentes: Y (luminancia o brillo), U (componente de crominancia azul), y V (componente de crominancia roja).

Donde:

1. **Y (Luminancia):**
- Representa el brillo de la imagen.
- Es similar a la imagen en escala de grises.
- La mayoría de la información sobre la estructura de la imagen se encuentra en el canal Y.
2. **U (Crominancia Azul):**
- Representa la información de color azul.
- Codifica la diferencia entre el canal azul y la luminancia.
3. **V (Crominancia Roja):**
- Representa la información de color rojo.
- Codifica la diferencia entre el canal rojo y la luminancia.

La separación de la luminancia y la crominancia permite un procesamiento más eficiente y efectivo, especialmente en tareas de procesamiento de imágenes y visión por computadora. YUV es más adecuado para compresión y transmisión de video porque la percepción humana es más sensible a los cambios en brillo (Y) que a los cambios en color (U y V). Esto permite comprimir los canales U y V más que el canal Y, ahorrando ancho de banda sin una pérdida perceptible de calidad.

Para tareas como la detección de líneas, trabajar en el espacio de color YUV puede simplificar el procesamiento. En lugar de trabajar con tres componentes RGB, se puede procesar principalmente el canal Y para detectar bordes y contrastes, reduciendo la carga computacional.

```
cam.size = OV7670_SIZE_DIV16
cam.colorspace = OV7670_COLOR_YUV
cam.flip_y = True
```

Imprimimos el ancho y alto de la imagen.

```
print(cam.width, cam.height)
```

Se crea un buffer para almacenar los datos de imagen capturados

```
buf = bytearray(2 * cam.width * cam.height)
print('##################################')
cam.capture(buf)
print(len(list(buf)))
```

Por medio del uso de caracteres le damos un nivel de intensidad distinto a cada parte de la imagen y se crea otro buffer que en este caso almacene una fila específica de los datos arrojados con la imagen

```
chars = b" .:-=+*#%@"
width = cam.width
row = bytearray(2 * width)
```

Configuramos el UART el cual es un receptor/transmisor asíncrono universal y define un protocolo, o conjunto de reglas para intercambiar datos en serie entre dos dispositivos por medio de dos cables y los valores para la normalización

```
# Configuración del UART
uart = busio.UART(board.GP16, board.GP17, baudrate=9600)
# Ajusta los pines y la velocidad según sea necesario

# Valores mínimos y máximos para normalización
desviacion_min = -195
desviacion_max = 195
```

Realizamos un ciclo “while” en el cual se captura una imagen y los datos se almacenan en un buffer y se itera sobre cada fila y columna de la misma

```
while True:
    cam.capture(buf)
    for j in range(cam.height):
        for i in range(cam.width):
            # Asegurar que el valor sea válido en el rango de un byte
            value = max(0, min(255, 225 - buf[2 * (width * j + i)]))
            row[i * 2] = row[i * 2 + 1] = value
        #print(list(row[0:40]))
```

Se realiza la suma de los valores de una de las líneas de la imagen para ser utilizada posteriormente

```
    mul = []

    for x in range(40): #Primeros valores de la fila
        y = row[x] * (x + 1) #Producto de la intensidad y la
posición
        mul.append(y)

    sum_mul = sum(mul)
    sum_cam = sum(row[0:40])
        print("sum_mul", sum_mul)
    print(sum_cam)
```

Se calcula la desviación de los datos proporcionados por la línea de imagen y se normaliza para que varien entre -100 cuando está al lado izquierdo y 100 cuando está muy al lado derecho, por ende cuando está completamente centrado sería un valor de 0.

En esta parte del código también se envían los datos por la conexión UART realizada anteriormente, convirtiéndolos de flotante a decimal para que sea más rápido enviar y recibir.

```
    if sum_cam != 0:
        p_medio = sum_mul / sum_cam
        print(p_medio)

        desviacion = (20.5 - p_medio) * 10
        print("desviacion", desviacion)

        # Normalizar la desviación
        desviacion_normalizada = ((desviacion - desviacion_min) /
(desviacion_max - desviacion_min)) * 200 - 100
        unu = int (desviacion_normalizada)
        print("desviacion_normalizada", unu)

        # Enviar la desviación normalizada por UART
        uart.write(f"{unu:.2f}\n".encode('utf-8'))
    else:
        # Enviar un valor de error o una desviación neutra si
sum_cam es 0
        uart.write("0.00\n".encode('utf-8'))

    print()
    #time.sleep(2)
```

En esta oportunidad se buscaba realizar el seguidor de línea por medio de códigos tradicionales y por medio de redes neuronales, para que este fuera capaz de entrenarse a seguir la línea por medio de refuerzo. Para ello se tomará en cuenta el código tradicional como código base y el de la red neuronal que en este caso es un perceptron se le harán ajustes para que funcione con ayuda de este.

Código tradicional: Seguidor de línea

Este código como se dijo anteriormente esta hecho en micro python en Raspberry Pi Pico W

En primer lugar se importan las librerías necesarias para manejar los pines, el PWM y la comunicación UART

```
from machine import Pin, PWM, UART
from time import sleep
```

Se configuran los pines del motor y se identifican entre motor A y B, adelante o atrás y se configura PWM en los pines para controlar su velocidad, se define la frecuencia y se toma una velocidad base

```
# Configuración de los pines del motor
Motor_A_Adelante = Pin(18, Pin.OUT)
Motor_A_Atras = Pin(19, Pin.OUT)
Motor_B_Adelante = Pin(20, Pin.OUT)
Motor_B_Atras = Pin(21, Pin.OUT)

# Configuración PWM para los motores
PWM_A_Adelante = PWM(Motor_A_Adelante)
PWM_A_Atras = PWM(Motor_A_Atras)
PWM_B_Adelante = PWM(Motor_B_Adelante)
PWM_B_Atras = PWM(Motor_B_Atras)

# Definir frecuencia PWM
pwm_frequency = 20000
PWM_A_Adelante.freq(pwm_frequency)
PWM_A_Atras.freq(pwm_frequency)
PWM_B_Adelante.freq(pwm_frequency)
PWM_B_Atras.freq(pwm_frequency)

# Velocidad base
velocidad_base = 360
```

Luego de esto se configura el UART

```
# Configuración del UART
uart = UART(0, baudrate=115200, tx=Pin(16), rx=Pin(17))
```

Creamos una función que ajusta la velocidad de un pin PWM, en la cual se ajusta la velocidad de 0 a 1023

```
def ajustar_velocidad(pwm_pin, velocidad):
    velocidad = max(0, min(int(velocidad), 1023))  # Convertir a
entero para pwm.duty_u16
    pwm_pin.duty_u16(velocidad * 64)
    print("Ajustar velocidad:", pwm_pin, "a", velocidad)
```

Se define una nueva función que en este caso detenga los motores

```
def detener_motores():
    # Detener ambos motores
    PWM_A_Adelante.duty_u16(0)
    PWM_A_Atras.duty_u16(0)
    PWM_B_Adelante.duty_u16(0)
    PWM_B_Atras.duty_u16(0)
    print("Motores detenidos")
```

Se crea una función para ajustar la velocidad de los motores basada en la desviación de la línea tomada de la imagen la cual calcula la velocidad de los motores A y B y asegura que esten entre 320 y 1023

```
def ajustar_motores(desviacion):
    # Calcula la velocidad ajustada para cada motor basado en la
desviación
    velocidad_a = velocidad_base - (desviacion * 2.5)
    velocidad_b = velocidad_base + (desviacion * 2.5)
```

```
    # Ajusta las velocidades dentro del rango permitido
    velocidad_a = max(320, min(velocidad_a, 1023))
    velocidad_b = max(320, min(velocidad_b, 1023))

    ajustar_velocidad(PWM_A_Adelante, velocidad_a)
    ajustar_velocidad(PWM_B_Adelante, velocidad_b)

    print(f"Desviación: {desviacion}, Velocidad A:
{velocidad_a}, Velocidad B: {velocidad_b}")
```

Por último se crea un bucle infinito "While" en el cual se comprueba si hay datos disponibles en el buffer UART

```
while True:
    if uart.any():
        try:
            # Leer una línea completa del buffer UART
            comando = uart.readline().decode('utf-8').strip()
            print(f"Comando recibido: {comando}")

            # Intentar convertir el comando a un número de punto
flotante
            desviacion = float(comando)
            ajustar_motores(desviacion)

            # Esperar un breve instante
            sleep(0.1)  # Tiempo de pausa, ajusta según tus
necesidades

            # Detener los motores
            detener_motores()


        except ValueError:
            print("Comando inválido recibido:", comando)

    #sleep(0.09)
```

Red neuronal: Perceptrón

El perceptrón es una red neuronal sencilla que toma una gran cantidad de datos con los cuales intenta predecir un resultado y lo compara con un resultado ya verificado para posteriormente corregir los pesos de la red neuronal y obtener una mejor predicción.

Los pesos mencionados anteriormente son el procedimiento para llegar a las predicciones, el valor de estos pesos se obtienen con la siguiente fórmula:

$$W_{i+1} = w_i + \alpha |\overline{Y}_c - \overline{Y}_r| \overline{X}^T$$

Donde $W_i$es el peso inicial, $\boldsymbol{\alpha}$ es el sesgo (controla qué tan predispuesta está la neurona a disparar un 1 o un 0), $\overline{Y}_r$ son los resultados verificados, $\overline{Y}_c$ son las predicciones y $\overline{X}^T$ son los datos de entrada transpuestos.

La predicción se calcula con la siguiente fórmula:

$$\overline{Y}_c = w_i \overline{X}$$

Para el código del perceptrón se utilizaron las clases "Matrix" y "Perceptron" tomadas de GitHub [1]

```
class Matrix:
    def __init__(self, m, n, data=None):
        self.m = m
        self.n = n
        if data is None:
            self.data = array.array('f', [0.0] * (n * m))
        else:
            if len(data) != n * m:
                raise ValueError("Incorrect data length")
            self.data = array.array('f', data)

    def __getitem__(self, index):
        if isinstance(index, tuple):
            i, j = index
            if isinstance(i, int) and isinstance(j, int):
                if 0 <= i < self.m and 0 <= j < self.n:
                    return self.data[i * self.n + j]
                else:
                    raise IndexError("Matrix indices out of range")
            if isinstance(i, slice) and isinstance(j, slice):
                start_i, stop_i, step_i = i.indices(self.m)
                start_j, stop_j, step_j = j.indices(self.n)
                sliced_data = [self.data[r * self.n + c] for r in
range(start_i, stop_i, step_i) for c in range(start_j, stop_j,
step_j)]
                return Matrix(stop_i - start_i, stop_j - stop_j,
sliced_data)
            else:
                raise IndexError("i,j indices are required")
        else:
            raise ValueError("i,j indices are required")

    def __setitem__(self, index, value):
        i, j = index
        if 0 <= i < self.m and 0 <= j < self.n:
            self.data[i * self.n + j] = value
        else:
            raise IndexError("Matrix indices out of range")

    def __add__(self, other):
        if isinstance(other, Matrix) and self.n == other.n and self.m
== other.m:
            result = Matrix(self.m, self.n)
            for i in range(self.m):
                for j in range(self.n):
                    result[i, j] = self[i, j] + other[i, j]
            return result
        else:
            raise ValueError("Matrices of different dimensions
cannot be added")

    def __sub__(self, other):
        if isinstance(other, Matrix) and self.n == other.n and self.m
== other.m:
            result = Matrix(self.m, self.n)
            for i in range(self.m):
                for j in range(self.n):
                    result[i, j] = self[i, j] - other[i, j]
            return result
        else:
            raise ValueError("Matrices of different dimensions
cannot be subtracted")
```

```
    def __mul__(self, other):
        if isinstance(other, (int, float)):
            result = Matrix(self.m, self.n)
            for i in range(self.m):
                for j in range(self.n):
                    result[i, j] = self[i, j] * other
            return result
        elif isinstance(other, Matrix):
            if self.n != other.m:
                raise ValueError("Number of columns of first matrix
must be equal to number of rows of second matrix")
            result = Matrix(self.m, other.n)
            for i in range(self.m):
                for j in range(other.n):
                    for k in range(self.n):
                        result[i, j] += self[i, k] * other[k, j]
            return result
        else:
            raise ValueError("Multiplication not defined for these
data types")

    def T(self):
        transposed_data = array.array('f', [0.0] * (self.n * self.m))
        for i in range(self.m):
            for j in range(self.n):
                transposed_data[j * self.m + i] = self.data[i * self.n +
j]
        return Matrix(self.n, self.m, transposed_data)

    def __or__(self, other):
        if isinstance(other, Matrix) and self.n == other.n:
            return Matrix(self.m + other.m, self.n, self.data +
other.data)
        else:
            raise ValueError("Matrices of different dimensions
cannot be added")

    def __and__(self, other):
        if self.m != other.m:
            raise ValueError("Matrices must have the same number
of rows to concatenate horizontally")

        concatenated_data = []
        for i in range(self.m):
            concatenated_data.extend(self.data[i*self.n :
(i+1)*self.n])
            concatenated_data.extend(other.data[i*other.n :
(i+1)*other.n])

        return Matrix(self.m, self.n + other.n, concatenated_data)

    def __str__(self):
        output = ""
        for i in range(self.m):
            row_str = " ".join(str(self[i, j]) for j in range(self.n))
            output += row_str + "\n"
        return output

class Perceptron:
    def __init__(self, input_size, output_size):
        self.weights = Matrix(input_size, output_size,
[random.random() for _ in range(input_size * output_size)])
        self.bias = Matrix(1, output_size, [random.random() for _
in range(output_size)])
```

```
    def predict(self, inputs):
        result = inputs * self.weights + self.bias
        return result

    def train(self, inputs, labels, learning_rate=0.1, epochs=1):
        for epoch in range(epochs):
            predictions = self.predict(inputs)
            error = labels - predictions
            self.weights += inputs.T() * error * learning_rate
            self.bias += error * learning_rate

    def save(self, filename):
        with open(filename, 'w') as f:
            data = {
                'weights': self.weights.data.tolist(),
                'bias': self.bias.data.tolist(),
                'input_size': self.weights.m,
                'output_size': self.weights.n
            }
            json.dump(data, f)

    def load(self, filename):
        with open(filename, 'r') as f:
            data = json.load(f)
            self.weights = Matrix(data['input_size'],
data['output_size'], data['weights'])
            self.bias = Matrix(1, data['output_size'], data['bias'])
```

La clase Matrix se utiliza para hacer operaciones con matrices y en la clase Perceptron se encuentran las fórmulas a seguir para el procedimiento de una red neuronal sencilla, además se le agregaron las funciones “save” y “load ” para guardar y cargar archivos con los datos de los pesos y las bias después de haber entrenado y así no perder el progreso.

En el resto del código simplemente implementamos las funciones al control de los motores para que el perceptrón tome los datos de la cámara y las velocidades resultantes para intentar predecir una mejor velocidad para los motores.

```
def ajustar_motores(perceptron, desviacion):
    entrada = Matrix(1, 1, [desviacion])
    salida = perceptron.predict(entrada)
    velocidad_a = 511 + salida[0, 0]
    velocidad_b = 511 - salida[0, 0]
    ajustar_velocidad(PWM_A_Adelante, velocidad_a)
    ajustar_velocidad(PWM_B_Adelante, velocidad_b)
    return velocidad_a, velocidad_b

uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))

perceptron = Perceptron(1, 1)
try:
    perceptron.load('perceptron_params.json')
    print("Parámetros cargados exitosamente.")
except FileNotFoundError:
    print("Archivo de parámetros no encontrado. Usando
parámetros aleatorios.")

inputs = Matrix(0, 1)  # Inicialmente vacío
labels = Matrix(0, 2)  # Inicialmente vacío

while True:
```

```
    if uart.any():
        comando = uart.read().decode('utf-8').strip()
        try:
            desviacion = float(comando)
            print(f"Comando recibido: {desviacion}")
            velocidad_a, velocidad_b = ajustar_motores(perceptron,
desviacion)

            # Agregar datos de entrenamiento
            inputs = inputs | Matrix(1, 1, [desviacion])
            labels = labels | Matrix(1, 2, [velocidad_a, velocidad_b])

            # Entrenar el perceptrón
            perceptron.train(inputs, labels, learning_rate=0.01,
epochs=1)

            # Guardar parámetros entrenados
            perceptron.save('perceptron_params.json')
        except ValueError:
            print(f"Comando inválido recibido: {comando}")
    sleep(0.1)
```

III. RESULTADOS

En este apartado se hará énfasis en las mejoras realizadas y los resultados obtenidos con estas.

Para que el carrito siguieran la línea correctamente tuvimos varios inconvenientes, que se solucionaron con mucha paciencia y mirando cada una de las cosas que pudieran interferir en ello, en primer lugar notamos que el carro no seguía la línea en diferentes entornos es decir, se le dificultaba el procesamiento en tiempo real de la imagen si no tenía la suficiente luz para detectar la línea a seguir por lo que la solución antes esto fue agregarle luz proveniente de una linterna para que siempre tuviera la misma luminosidad y no interfiriera en los datos arrojados.

Una mejora un poco más a nivel de montaje fue que los motores que vienen incluidos al chasis por alguna razón no funcionaban de manera correcta y al cambiarlos se soluciono, pero se pudo evidenciar otro problema y era que estos nuevos motores requieren mayor corriente por lo que hubo que agregarle otra pila de 9V al montaje, es decir que que quedó con dos pilas de 9V en paralelo para que las ruedas pudiesen girar bien, sin detenerse o trabarse, luego de ello el problema quedó resuelto. La configuración PWM de los motores y el ajuste dinámico de la velocidad basado en la desviación calculada permitieron un control preciso del movimiento del carrito. Este control preciso permitió al carrito seguir la línea con exactitud, incluso en curvas levemente pronunciadas y cambios en la trayectoria.

En cuanto a la comunicación UART entre las Raspberry también tuvimos problemas dado que los datos no llegaban de forma correcta ni rápidamente, esto se soluciono convirtiendo los datos flotantes resultantes de las operaciones en datos enteros lo cual hizo que se redujera el tiempo de envio puesto que los bits que se requieren son menos. La lectura y procesamiento de los datos de desviación a través de UART en tiempo real permitió al carrito responder rápidamente a cualquier desviación de la línea, ajustando las velocidades de los motores en consecuencia. Esto resultó en un seguimiento de línea fluido y continuo.

Por último en las mejoras que se hicieron se normalizaron los datos que enviaba la cámara dado que antes de eso no estaban en un rango específico sino que cada vez que se probaba el carrito en movimiento variaba, lo que no permitía hacer un movimiento correcto.

IV.CONCLUSIONES

- La cámara permite capturar imágenes de alta resolución que pueden ser procesadas para detectar líneas de diferentes formas y colores, pero a su vez el procesamiento de imágenes en tiempo real es más complejo y requiere más recursos que el uso de sensores simples.
- El uso del espacio de color YUV simplificó el procesamiento de imágenes, permitiendo una detección más rápida y precisa de la línea. La separación de luminancia y crominancia ayudó a enfocar el procesamiento en la estructura de la imagen (canal Y), ignorando los detalles de color que no eran relevantes para la detección de la línea.
- El procesamiento eficiente en la Raspberry Pi Pico W, con una utilización optimizada de recursos de memoria y CPU, demuestra que los microcontroladores de bajo costo pueden manejar tareas complejas como el procesamiento de imágenes en tiempo real y el control de motores simultáneamente.
- La elección de componentes como la cámara OV7670 y la Raspberry Pi Pico W proporciona una solución coste-efectiva sin sacrificar el rendimiento. Esto hace que el proyecto sea accesible para entusiastas y estudiantes con presupuestos limitados, al tiempo que ofrece capacidades robustas.
- Este proyecto sirve como una excelente herramienta educativa, permitiendo a los estudiantes y aficionados aprender sobre visión por computadora, procesamiento de señales, control de motores y comunicación serial. La práctica con este tipo de proyectos puede desarrollar habilidades prácticas y teóricas valiosas.

REFERENCIAS

[1] G. Muñoz. “GerardoMunoz - Overview”. GitHub. Accedido el 5 de junio de 2024. [En línea]. Disponible: https://github.com/GerardoMunoz

[2] Omnivision. “OV7670_DS”. MIT - Massachusetts Institute of Technology. Accedido el 5 de junio de 2024. [En línea]. Disponible: https://web.mit.edu/6.111/www/f2016/tools/OV7670_2006.pdf