

Robot Seguidor de línea

Microcontroladores

Docente: Gerardo Munoz

Adrian Elias Causil Villadiego

Departamento de Ingeniería

Proyecto curricular de Ingeniería Electrónica

Bogotá, Colombia

aecausilv@udistrital.edu.co

Joel Eduardo Reyes

Departamento de Ingeniería

Proyecto curricular de Ingeniería Electrónica

Bogotá, Colombia

correo

Nicolas Felipe Macias Paz

Departamento de Ingeniería

Proyecto curricular de Ingeniería Electrónica

Bogotá, Colombia

nfmaciasp@udistrital.edu.co

Resumen—Este informe presenta el desarrollo e implementación de un seguidor de línea basado en una lógica de control con redes neuronales, utilizando raspberry pi pico W y una cámara OV767 para la recolección de datos. Inicialmente basando el sistema a partir de un perceptrón, un tipo de red neuronal simple, y consiguiente a este, mejorando dicho sistema por medio de una red definida a través del aprendizaje por refuerzo. Se detalla la implementación del seguidor con una pantalla LSD para la visualización de la pista y el control de los motores mediante un puente H L298.

Justificando la implementación de una segunda red neuronal, se reconocen las limitaciones del perceptrón en términos de adaptabilidad y precisión al ser una red neuronal simple, por esto, se implementa una red neuronal basada en el aprendizaje por refuerzo. En donde se evidencia una mejora en su desempeño mediante la experiencia, ajustando el comportamiento del seguidor para que responda de forma óptima a las variaciones que presente la pista.

Los resultados experimentales demuestran que la integración del aprendizaje por refuerzo mejora significativamente la capacidad del robot para seguir la línea de manera precisa y adaptativa, en comparación con la red neuronal basada en el perceptrón. Este enfoque no solo proporciona un control más robusto y eficiente, sino que también abre nuevas posibilidades para la optimización del tiempo en el que el robot puede completar la pista.

Index Terms—Aprendizaje por refuerzo, L298, LSD, OV7670, Perceptrón, Red neuronal, Robot, Seguidor de línea.

Abstract—This report presents the development and implementation of a line tracker based on a logic of control with neural networks, using raspberry pi peak W and an OV767 camera for data collection. Initially basing the system from a perceptron, a type of simple neural network, and consequent to it, improving the system through a network defined through reinforcement learning. It

details the implementation of the tracker with an LSD display for the display of the tread and the control of the motors by means of a bridge H L298.

Justifying the implementation of a second neural network, perceptron limitations are recognized in terms of adaptability and accuracy being a simple neural network, therefore, a neural network based on learning by reinforcement is implemented. Where there is evidence of an improvement in their performance through experience, adjusting the behavior of the follower to respond optimally to variations presented by the track.

Experimental results show that the integration of reinforcement learning significantly improves the robot's ability to follow the line in a precise and adaptive way, compared to the perceptron-based neural network. This approach not only provides more robust and efficient control, but also opens up new possibilities for optimizing the time in which the robot can complete the track.

Index Terms—Reinforcement learning, L298, LSD, OV7670, Perceptron, Neural network, Robot, Line follower.

I. INTRODUCCIÓN

La robótica móvil es un área que se encuentra en constante evolución, impulsada por los avances que permiten la creación de sistemas más inteligentes y flexibles, como lo pueden ser las redes neuronales.

En la última década, los avances en la inteligencia artificial y la robótica han permitido desarrollar sistemas autónomos cada vez más sofisticados. Uno de los campos de aplicación más interesantes es el de los robots seguidores de línea, que tienen una amplia gama de usos en la industria, la logística y la investigación académica. Estos robots utilizan sensores para detectar y seguir una línea trazada en el suelo, realizando tareas como el transporte de objetos o la recolección de datos

en entornos estructurados.

El presente informe describe el desarrollo de un robot seguidor de línea basado en una Raspberry Pi Pico W, una cámara OV7670 y un puente H para el control de los motores. El sistema de control del robot se basa inicialmente en un perceptrón, una de las formas más simples de red neuronal, para procesar las señales de la cámara y generar comandos de movimiento. Posteriormente, se implementa una red neuronal más compleja utilizando aprendizaje por refuerzo, lo que permite al robot mejorar su desempeño a través de la experiencia y la interacción con su entorno.

El objetivo principal de este proyecto es diseñar y construir un robot capaz de seguir una línea de manera eficiente y adaptativa, utilizando técnicas avanzadas de inteligencia artificial. Se espera que la implementación de un sistema de control basado en redes neuronales permita al robot manejar mejor las variaciones en las condiciones del entorno y superar las limitaciones de los métodos tradicionales de seguimiento de líneas.

II. MARCO TEÓRICO

II-A. Raspberry pi pico W

La Raspberry Pi, una microcomputadora de placa única equipada con procesadores ARM, es ampliamente utilizada en proyectos de microcontroladores avanzados como los seguidores de línea. Utiliza Python como su principal lenguaje de programación, lo que facilita la implementación de algoritmos complejos y la integración de múltiples sensores y actuadores. A diferencia de otros microcontroladores tradicionales, como los de la serie Arduino, la Raspberry Pi ofrece mayor capacidad de procesamiento y flexibilidad gracias a su capacidad para ejecutar un sistema operativo completo y aplicaciones más sofisticadas. Esto la convierte en una opción ideal para proyectos que requieren mayor potencia de cálculo y versatilidad.

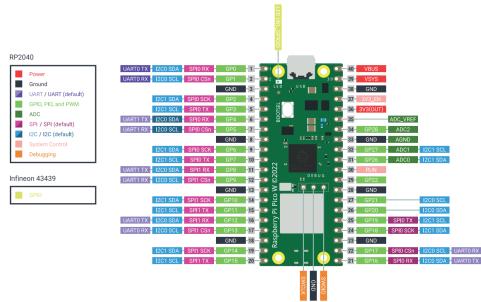


Figura 1. Rasp Berry pi PicoW

El RP2040, desarrollado por la Fundación Raspberry Pi y utilizado en la Raspberry Pi Pico, es un microcontrolador con un procesador de doble núcleo ARM Cortex-M0+ que opera hasta 133 MHz, complementado con 264 KB de RAM y 2 MB de memoria flash QSPI. Cuenta con 30 pines GPIO, 4 de ellos capaces de funcionar como entradas analógicas, y soporta

periféricos como 2 UART, 2 SPI, 2 I2C, 16 canales PWM, un controlador USB 1.1 y 8 PIO para manejar protocolos personalizados. Aunque no tiene conectividad inalámbrica integrada, su arquitectura incluye un cristal de 12 MHz, RTC, temporizadores integrados, soporte para depuración SWD y un controlador DMA. Además, su rango de voltaje operativo de 1.8V a 3.3V y su bajo consumo de energía lo hacen ideal para una amplia gama de aplicaciones embebidas, desde proyectos de hobby hasta implementaciones industriales.

II-B. Redes neuronales

Las redes neuronales artificiales (RNA) son modelos computacionales inspirados en el funcionamiento del cerebro humano. Están diseñadas para reconocer patrones complejos y tomar decisiones basadas en esos patrones. Se componen de nodos (neuronas) organizados en capas (entrada, ocultas y salida), conectados entre sí mediante pesos sinápticos.

El perceptrón es una de las formas más simples de red neuronal y sirve como base para redes más complejas. Fue introducido por Frank Rosenblatt en 1958 y consiste en una sola capa de neuronas de salida directamente conectadas a una capa de entrada. Cada conexión tiene un peso asociado y la neurona de salida realiza una suma ponderada de las entradas, seguida por una función de activación (normalmente una función escalón o sigmoide).

Las entradas x_1, x_2, \dots, x_n son las señales de entrada, los pesos w_1, w_2, \dots, w_n son los coeficientes asociados, y la función de activación $f(z)$ se aplica a la suma ponderada:

$$z = \sum_{i=1}^n w_i x_i + b \quad (1)$$

donde b es el sesgo. La salida del perceptrón es $y = f(z)$.

El aprendizaje por refuerzo (RL, por sus siglas en inglés) es una técnica donde un agente aprende a tomar decisiones mediante interacciones con un entorno. El objetivo es maximizar una señal de recompensa acumulativa. Las redes neuronales en RL se utilizan para aproximar funciones de valor o políticas que guían al agente.

La política (π) es la estrategia que utiliza el agente para decidir acciones basadas en estados. La función de valor ($V(s)$) representa el valor esperado de las recompensas futuras desde el estado s , mientras que la función $Q(s, a)$ denota el valor esperado de realizar la acción a en el estado s .

Las redes neuronales profundas (DNN) han mejorado significativamente las capacidades de RL, permitiendo el uso de aproximaciones de función más complejas y eficientes, como en el caso del algoritmo Deep Q-Learning.

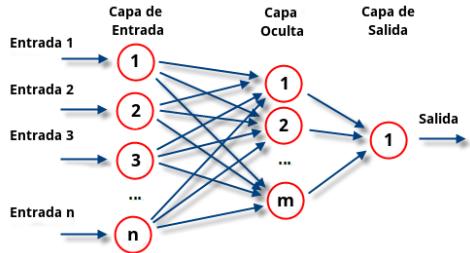


Figura 2. Redes neuronales

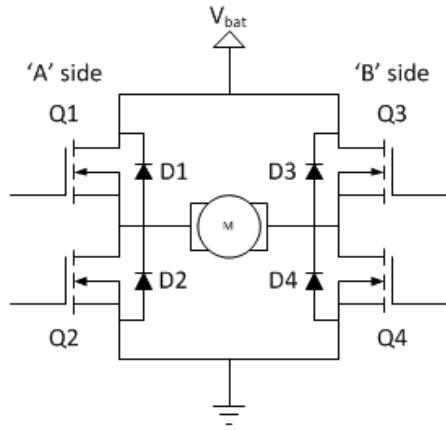


Figura 3. Puente H

II-C. Puente H

III. RESULTADOS

Un puente en H es un circuito integrado que se utiliza en robótica y en muchas otras aplicaciones para aplicar una cantidad ajustable de potencia a una carga, normalmente un motor de corriente continua. Es un circuito bastante sencillo compuesto por transistores colocados en una configuración en forma de H con la carga en el centro, de ahí el nombre de puente en H. Cada transistor está emparejado con un diodo flyback que evita que se dañe el transistor durante la polarización inversa. El número de transistores y diodos de un puente H depende del tipo de motor que controla, ya sea monofásico o trifásico. Aunque es sencillo, el puente H desempeña un papel crucial en el control de motores, ya que nos permite accionar un motor de forma bidireccional [1].

El puente H para motores monofásicos utiliza cuatro transistores para controlar la dirección de la corriente que circula por el motor. Para que el motor gire hacia delante o hacia atrás, sólo se pueden encender dos transistores a la vez. Los transistores utilizados en un puente H suelen ser transistores bipolares o FET, como los MOSFET y los IGBT. Los elementos de conmutación pueden encenderse y apagarse de forma independiente, pero sólo hay unas pocas configuraciones específicas que harán funcionar el motor de forma eficaz sin cortocircuitarlo.

Si los transistores Q1 y Q4 están encendidos como se muestra en la imagen, el motor girará en sentido contrario a las agujas del reloj. Por el contrario, si los transistores Q3 y Q2 están encendidos, el motor girará en el sentido de las agujas del reloj. Como un transistor de cada lado del puente está siempre encendido, siempre habrá un camino continuo para que la corriente fluya a través del motor.

En la presente sección, se presentan los datos obtenidos durante las pruebas del seguidor de línea. Se realizaron mediciones detalladas de corriente, voltaje y potencia; se evaluó también el rendimiento del sistema bajo diferentes condiciones de iluminación. Estos parámetros son cruciales para entender el comportamiento y la eficiencia del robot, así como para identificar posibles mejoras en su diseño y operación.

A continuación se muestran las tablas obtenidas con las mediciones de ambos motores para el sentido horario y antihorario respectivamente, las tomas de estas mediciones se encuentran en anexos.

Motor derecho		
Corriente (mA)	Voltaje (V)	Potencia (mW)
126,03	7,568	953,79504
115,19	7,459	859,20221

Motor izquierdo		
Corriente (mA)	Voltaje (V)	Potencia (mW)
121,26	7,533	913,45158
116,66	7,336	855,81776

Se midió el rango de funcionamiento lumínico del carro utilizando un luxómetro, lo que permitió determinar cómo diferentes niveles de luz afectan su rendimiento. Los datos recopilados se representaron en una gráfica para visualizar la relación entre la intensidad lumínica y la eficacia del seguidor de línea, proporcionando una comprensión más profunda de su desempeño en diversas condiciones de iluminación.

Nivel de Funcionamiento %

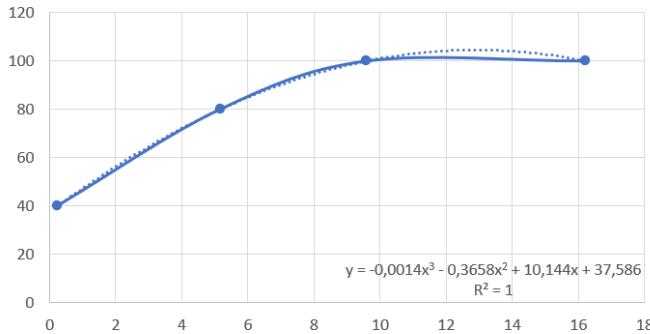


Figura 4. Prueba luminidad del ambiente

Finalmente, el seguidor de línea fue probado en la pista sugerida por el docente, donde demostró un desempeño sobresaliente al no salirse de la línea en ningún momento. Este resultado confirma que el sistema cumple con el objetivo establecido, mostrando una capacidad confiable y eficiente para seguir una trayectoria predeterminada con precisión.

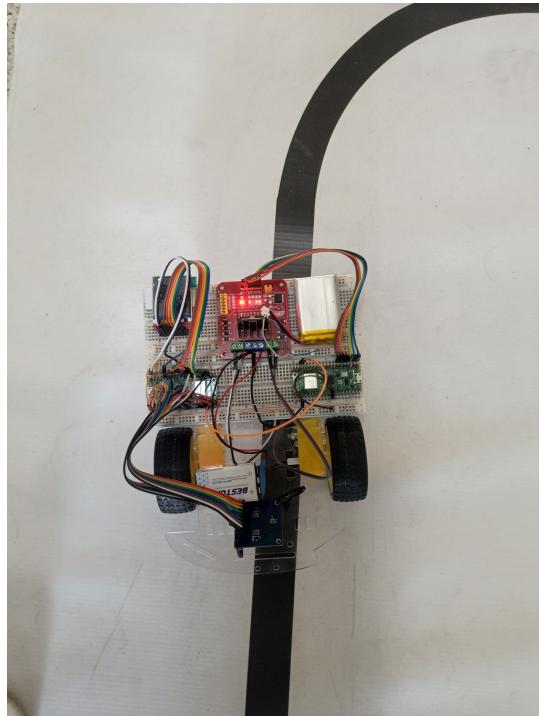


Figura 5. Estructura Física

IV. CÓDIGO

En esta sección, se presenta y explica el código utilizado para el control del robot. El código está escrito en Python y emplea diversas bibliotecas y clases para gestionar la comunicación UART, la configuración de pines, el control de motores mediante un controlador L298N y la implementación de un perceptrón simple para la toma de decisiones basada en los datos recibidos. El sistema recibe datos en tiempo real, ajusta las velocidades de los motores en función de estos datos y puede entrenar o predecir los comportamientos del motor según la posición de un interruptor. A continuación, se detallan las partes principales del código y su funcionamiento.

```

1 import network, socket, time
2 from machine import Timer, Pin, PWM, UART
3 from L298N_motor import L298N
4 from Matrix import Matrix
5 import array
6 import random
7 uart = UART(0, baudrate=115200, bits=8,
8     parity=None, stop=1, rx=Pin(17), tx=
9     Pin(16))
10 ENA = PWM(Pin(0))
11 IN1 = Pin(1, Pin.OUT)
12 IN2 = Pin(2, Pin.OUT)
13 IN3 = Pin(3, Pin.OUT)
14 IN4 = Pin(4, Pin.OUT)
15 ENB = PWM(Pin(5))
16 SWITCH_PIN = 27
17 switch_pin = Pin(SWITCH_PIN, Pin.IN, Pin.
18     PULL_UP)

```

Estas líneas importan las bibliotecas necesarias para la red, la comunicación por socket, el manejo del tiempo, el uso de pines, PWM (modulación por ancho de pulso), UART (comunicación serial), y la manipulación de matrices y números aleatorios. Posteriormente se configura la comunicación UART con los pines especificados y se configuran los pines para controlar los motores. ENA y ENB son PWM para controlar la velocidad de los motores, mientras que IN1 a IN4 son pines de control de dirección. switchpin es un pin de entrada con una resistencia pull-up.

```

1 motor1 = L298N(ENA, IN1, IN2)
2 motor2 = L298N(ENB, IN3, IN4)
3 class Perceptron:
4     def __init__(self, name_or_input_size
5         , output_size=None):
6         if isinstance(name_or_input_size,
7             str):
7             self.weights = Matrix.
8                 load_file(
9                     name_or_input_size)
10        else:
11            self.weights = Matrix(
12                name_or_input_size + 1,
13                output_size + 1, [random.
14                    random() for _ in range((
15                        name_or_input_size + 1) *
16                        (output_size + 1))])

```

```

1 def predict(self, inputs):
2     tail = False
3     if inputs.n == self.weights.m -
4         1:
5         result = Matrix.untail(Matrix
6             .tail(inputs) * self.
7             weights)
8     else:
9         result = inputs * self.
10            weights
11     return result
12
13 def train(self, inputs, labels,
14           learning_rate=0.01, epochs=1):
15     if inputs.n == self.weights.m -
16         1:
17         inputs = Matrix.tail(inputs)
18     if labels.n == self.weights.n -
19         1:
20         labels = Matrix.tail(labels)
21     for epoch in range(epochs):
22         predictions = self.predict(
23             inputs)
24         error = labels - predictions
25         self.weights = self.weights.
26             add_tail(inputs.T() *
27             error * learning_rate)
28
29 def save_file(self, name):
30     self.weights.save_file(name)

```

La clase ‘Perceptron’ implementa un modelo de red neuronal simple, utilizado para tareas de clasificación binaria. Su método de inicialización permite dos modos: cargar los pesos desde un archivo o inicializarlos aleatoriamente para una estructura dada. El método ‘predict’ toma una matriz de entradas y realiza una predicción multiplicándolas por la matriz de pesos, ajustando las entradas para incluir el sesgo si es necesario. El método ‘train’ ajusta los pesos del perceptrón para minimizar el error entre las predicciones y las etiquetas proporcionadas, realizando varias épocas de entrenamiento y aplicando la regla de aprendizaje del perceptrón. Además, el método ‘savefile’ permite guardar la matriz de pesos en un archivo para su posterior uso. En el contexto del código, la clase ‘Perceptron’ se utiliza para ajustar dinámicamente las velocidades de los motores basándose en las lecturas de sensores o datos recibidos a través de UART, permitiendo que el sistema controle los motores de manera efectiva según las condiciones actuales y el entrenamiento previo.

```

1 if __name__ == "__main__":
2     val = ""
3     datoant = None
4
5     perceptron = Perceptron(
6         name_or_input_size=1, output_size
7         =2)
8     def load():
9         global perceptron
10        perceptron = Perceptron("m.txt")

```

En esta sección del código se inicializa el perceptrón y

las variables globales además que se cargan los pesos del perceptrón desde el archivo m.txt, la cual es a matriz que se obtuvo al entrenarlo y es la que mejor respuesta arroja en el seguimiento de la linea, el archivo txt es el que se ve a continuación

1	2	3
2	-0.1880123	0.1716713 1.0
3	0.8826846	0.8131949 1.0

```

1 def train_control():
2     global cont_train, repeat, val
3     if repeat:
4         dat = int(val)
5         dat1 = int(val) / 100
6         floor_colors = Matrix(1, 1, [dat1])
7         if train:
8             global datoant
9             stop_timer = Timer(-1)
10
11         def stop_motors():
12             motor1.stop()
13             motor2.stop()
14
15         stop_timer.deinit()
16         datoant = 0
17
18         if dat == -datoant:
19             return
20         else:
21             datoant = dat
22             if -30 <= dat <= 30:
23                 motor1.setSpeed(59000)
24                 motor2.setSpeed(56000)
25                 v_i = (59000 / 65000)
26                 v_d = (56000 / 65000)
27                 motors_vel = Matrix(1, 2,
28                     [v_i, v_d])
29                 motor1.forward()
30                 motor2.forward()
31                 stop_timer.init(period
32                     =35, mode=Timer.
33                     ONE_SHOT, callback=
34                     lambda t: stop_motors
35                     ())
36
37         elif -55 <= dat < -30:
38             motor1.setSpeed(61000)
39             motor2.setSpeed(45000)
40             v_i = (61000 / 65000)
41             v_d = (45000 / 65000)
42             motors_vel = Matrix(1, 2,
43                 [v_i, v_d])
44             motor1.forward()
45             motor2.forward()
46             stop_timer.init(period
47                 =27, mode=Timer.
48                 ONE_SHOT, callback=
49                 lambda t: stop_motors
50                 ())

```

```

1 elif dat < -55:
2     motor1.setSpeed(65000)
3     motor2.setSpeed(45000)
4     v_i = (65000 / 65000)
5     v_d = (45000 / 65000)
6     motors_vel = Matrix(1, 2,
7         [v_i, v_d])
8     motor1.forward()
9     motor2.forward()
10    stop_timer.init(period
11        =27, mode=Timer.
12        ONE_SHOT, callback=
13            lambda t: stop_motors
14                ())
15
16 elif 30 < dat <= 62:
17     motor1.setSpeed(46000)
18     motor2.setSpeed(55000)
19     v_i = (46000 / 65000)
20     v_d = (55000 / 65000)
21     motors_vel = Matrix(1, 2,
22         [v_i, v_d])
23     motor1.forward()
24     motor2.forward()
25     stop_timer.init(period
26         =25, mode=Timer.
27        ONE_SHOT, callback=
28            lambda t: stop_motors
29                ())
30
31 elif dat > 62:
32     motor1.setSpeed(46000)
33     motor2.setSpeed(59000)
34     v_i = (46000 / 65000)
35     v_d = (59000 / 65000)
36     motors_vel = Matrix(1, 2,
37         [v_i, v_d])
38     motor1.forward()
39     motor2.forward()
40     stop_timer.init(period
41         =25, mode=Timer.
42        ONE_SHOT, callback=
43            lambda t: stop_motors
44                ())
45
46 else:
47     stop_timer = Timer(-1)
48
49     stop_timer.deinit()
50
51 def stop_motors():
52     motor1.stop()
53     motor2.stop()
54
55     prediction = perceptron.predict(
56         floor_colors)
57     v_i = prediction[0, 0]
58     v_d = prediction[0, 1]
59     print(v_i, v_d)
60     m1 = abs(int(v_i * 65000))
61     m2 = abs(int(v_d * 65000))
62     motor1.setSpeed(m1)
63     motor2.setSpeed(m2)
64     motor1.forward()
65     motor2.forward()
66     stop_timer.init(period=34, mode=
67         Timer.ONE_SHOT, callback=
68             lambda t: stop_motors())

```

La función traincontrol es fundamental para el control dinámico de los motores en función de los datos recibidos. Esta función primero convierte los datos de entrada en un formato adecuado para el perceptrón. Si el sistema está en modo de entrenamiento, ajusta las velocidades de los motores y los mueve en la dirección apropiada basada en el valor de entrada, entrenando simultáneamente el perceptrón con las velocidades calculadas. Dependiendo del valor de los datos, ajusta las velocidades y las direcciones de los motores para manejar diversas condiciones de movimiento. Si no está en modo de entrenamiento, utiliza el perceptrón previamente entrenado para predecir las velocidades de los motores en función de las entradas actuales, ajustando los motores en consecuencia. La función también incluye mecanismos de temporización para detener los motores después de un periodo específico, asegurando que el movimiento sea controlado y preciso.

```

1 while True:
2     repeat = True
3     if uart.any():
4         rd = uart.read(3)
5         if rd != None:
6             val = int(rd.decode('utf
7                 -8', 'replace'))
8
9             if switch_pin.value() ==
10                 0:
11                 train = True
12             else:
13                 train = False
14                 load()
15
16             train_control()

```

El bucle principal continuamente verifica si hay datos disponibles en el UART. Si los hay, los lee y decide si entrenar el perceptrón o predecir basándose en la posición del interruptor. Luego llama a traincontrol para actuar en consecuencia.

Ahora se va a explicar el código usado en la Raspberry con circuit python, la cual se encarga de la visualización con la cámara y mandar el valor de la desviación por UART

```

1 import sys
2 import time
3 import digitalio
4 import busio
5 import board
6 from adafruit_st7735r import ST7735R
7 import displayio
8 import terminalio
9 from adafruit_display_text import label
10 from collections import deque
11 from adafruit_ov7670 import OV7670,
12     OV7670_SIZE_DIV16, OV7670_COLOR_YUV

```

```
1 uart = busio.UART(board.GP16, board.GP17,
2                     baudrate=115200, bits=8, parity=None,
3                     stop=2)
4 bufw = bytearray(10)
5 mosi_pin = board.GP19
6 clk_pin = board.GP18
7 reset_pin = board.GP22
8 cs_pin = board.GP26
9 dc_pin = board.GP28
10 displayio.release_displays()
```

Estas líneas importan las bibliotecas necesarias para la comunicación UART, la configuración de pines, el manejo de la pantalla y la cámara y se configura la comunicación UART con los pines y parámetros especificados. También se definen los pines necesarios para la comunicación SPI con la pantalla.

```
1 spi = busio.SPI(clock=clk_pin, MOSI=mosi_pin)
2 display_bus = displayio.FourWire(spi,
3                                 command=dc_pin, chip_select=cs_pin,
4                                 reset=reset_pin)
5 display = ST7735R(display_bus, width=128,
6                     height=160, bgr=True)
7 splash = displayio.Group()
8 display.show(splash)
9
10 color_bitmap = displayio.Bitmap(128, 160,
11                                1)
12 color_palette = displayio.Palette(1)
13 color_palette[0] = 0x00FF00
14 bg_sprite = displayio.TileGrid(
15     color_bitmap, pixel_shader=
16     color_palette, x=0, y=0)
17 splash.append(bg_sprite)
18
19 inner_bitmap = displayio.Bitmap(125, 155,
20                                1)
21 inner_palette = displayio.Palette(1)
22 inner_palette[0] = 0x000000
23 inner_sprite = displayio.TileGrid(
24     inner_bitmap, pixel_shader=
25     inner_palette, x=5, y=5)
26 splash.append(inner_sprite)
```

El código configura una pantalla ST7735R utilizando comunicación SPI. Inicialmente, se definen los pines necesarios para SPI, incluyendo ‘mosi_pin’, ‘clk_pin’, ‘reset_pin’, ‘cs_pin’, y ‘dc_pin’. Luego, se crea una interfaz de bus SPI de cuatro hilos (‘displayio.FourWire’) con estos pines, y se inicializa la pantalla con dimensiones de 128x160 píxeles en formato de color BGR. Se crea un grupo principal ‘splash’ para contener los elementos gráficos, que se muestra en la pantalla. Se define un bitmap de fondo verde (‘color_bitmap’) y una paleta de un solo color verde brillante, que se añaden al grupo ‘splash’ como un sprite (‘bg_sprite’). Además, se dibuja un rectángulo negro interior (‘inner_bitmap’) con su propia paleta y se posiciona dentro del grupo ‘splash’ con un borde de 5 píxeles. Esta configuración permite visualizar gráficos y texto en la pantalla ST7735R, proporcionando una interfaz visual clara y efectiva para el proyecto.

```

1 cam_bus = busio.I2C(board.GP21, board.
2     GP20)
3 cam = OV7670(
4     cam_bus,
5     data_pins=[
6         board.GP0, board.GP1, board.GP2,
7         board.GP3,
8         board.GP4, board.GP5, board.GP6,
9         board.GP7
10    ],
11    clock=board.GP8, vsync=board.GP13,
12    href=board.GP12,
13    mclk=board.GP9, shutdown=board.GP15,
14    reset=board.GP14
15 )
16 cam.size = OV7670_SIZE_DIV16
17 cam.colorspace = OV7670_COLOR_YUV
18 cam.flip_y = True
19 buf = bytearray(2 * cam.width * cam.
20     height)

```

El código configura una cámara OV7670 para capturar imágenes y procesar datos. Primero, se definen los pines para la comunicación I2C ('cam_bus') y los pines específicos de la cámara, incluyendo 'data_pins' para las líneas de datos, 'clock' para el reloj, 'vsync' y 'href' para la sincronización vertical y horizontal, 'mclk' para el reloj maestro, 'shutdown' y 'reset' para el control de energía y reinicio de la cámara. La cámara se inicializa con estas configuraciones y se ajusta su tamaño de imagen a 'OV7670_SIZE_DIV16' y su espacio de color a 'OV7670_COLOR_YUV'. Además, se activa la inversión vertical de la imagen ('cam.flip_y = True'). Un buffer ('buf') se prepara para almacenar los datos de imagen capturados. Durante la captura, la cámara almacena los datos de imagen en el buffer, que luego se procesan para calcular la desviación en función de una lista predefinida, comparando los bytes de las imágenes capturadas. La desviación se envía a través de UART, permitiendo la comunicación de los resultados al sistema principal. Esta configuración y procesamiento de la cámara OV7670 permiten capturar y analizar imágenes en tiempo real.

```

1 combined_bitmap = displayio.Bitmap(128,
2     24, 3)
3 combined_palette = displayio.Palette(3)
4 combined_palette[0] = 0x000000
5 combined_palette[1] = 0x0000FF
6 combined_palette[2] = 0xFF0000
7 combined_sprite = displayio.TileGrid(
8     combined_bitmap, pixel_shader=
9         combined_palette, x=0, y=68)
10 splash.append(combined_sprite)
11
12 text_group = displayio.Group(scale=2, x
13     =11, y=25)
14 text_area = label.Label(terminalio.FONT,
15     text="", color=0xFFFFF)
16 text_group.append(text_area)
17 splash.append(text_group)

```

El código configura un sistema para capturar y visualizar datos de una cámara OV7670 en una pantalla. Se crea un búfer de bytes ('buf') para almacenar los datos de la imagen, y se define una cadena ('chars') para representar diferentes niveles de intensidad de los píxeles. Se establece una lista predefinida ('predefined_list') para comparar con los datos capturados y tres variables ('last_rows') para almacenar las últimas tres filas de datos de la imagen.

Se configura un bitmap ('combined_bitmap') de 128x24 píxeles con una paleta de tres colores para mostrar las filas combinadas de datos en la pantalla, y se crea un sprite de mosaico ('combined_sprite') para visualizarlo en las coordenadas (0, 68). Además, se crea un grupo de texto ('text_group') con un área de texto ('text_area') para mostrar información en la pantalla, posicionado en las coordenadas (11, 25). Todos estos elementos se añaden al grupo principal de la pantalla ('splash').

```

1 def calcular_desviacion(bytarray_1,
2     bytarray_2, bytarray_3,
3     predefined_list):
4     diferencias = sum(1 for i in range(
5         len(predefined_list)) if
6         bytarray_1[i] != predefined_list[
7             i] or bytarray_2[i] !=
8             predefined_list[i] or bytarray_3[
9                 i] != predefined_list[i])
10    desviacion_porcentaje = (diferencias
11        / len(predefined_list)) * 100
12    if bytarray_1 < predefined_list:
13        desviacion_porcentaje *= -1
14    elif bytarray_1 > predefined_list:
15        desviacion_porcentaje *= 1
16    return desviacion_porcentaje

```

Esta función compara tres matrices de bytes con una lista predefinida y calcula la desviación en porcentaje, indicando si la desviación es positiva o negativa.

```

1 while True:
2     desv = 100
3     cam.capture(buf)
4     pant_row = []
5     last_rows[2] = last_rows[1]
6     last_rows[1] = last_rows[0]
7     last_rows[0] = bytarray(row)
8     if last_rows[0] is not None and
9         last_rows[1] is not None and
10        last_rows[2] is not None:
11            desv = calcular_desviacion(
12                last_rows[2], last_rows[1],
13                last_rows[0], predefined_list)
14
15     text = ""
16     for j in range(cam.height - 3, cam.
17         height):
18         for i in range(cam.width):
19             row[i * 2] = row[i * 2 + 1] =
20                 chars[buf[2 * (width * j
21                     + i)] * (len(chars) - 1)
22                     // 255]
23     msg = str(int(desv))
24     if msg is not None:
25         data = msg.encode(5)
26         uart.write(data)
27
28     text += row.decode() + "\n"
29     text_area.text = text

```

El bucle principal captura la imagen de la cámara, actualiza las matrices de bytes con las últimas filas de píxeles, calcula la desviación y muestra el resultado en la pantalla. Si hay una desviación calculada, la envía a través de UART.

V. CONSIDERACIONES FINALES

Durante el desarrollo del proyecto, se encontraron varios desafíos que requirieron soluciones específicas. Inicialmente, la lentitud de la cámara al revisar todo el rango de visión se solucionó limitando la captura y análisis a las últimas tres filas de datos. La rapidez del UART también fue un problema; para solucionarlo, se cambiaron los jumpers por cables UTP y se aumentó el baudrate hasta el límite permitido por el cable. El desnivel en la velocidad de los motores se abordó mediante pruebas exhaustivas para determinar un funcionamiento óptimo cuando los motores operan en paralelo. El control automático para el entrenamiento presentó dificultades debido al retardo en el envío de información por UART y el correspondiente accionamiento. Esto se solucionó creando una función que detiene los motores después de cada lectura para permitir un siguiente dato correcto. El análisis de datos para la red neuronal requería normalización, por lo que se ajustaron tanto la desviación como las velocidades de los motores para asegurar el correcto funcionamiento de la red. Finalmente, para permitir que la pantalla mostrara una imagen de lo que veía el carro, se realizó una transformación necesaria debido a la naturaleza de la pantalla SPI. Estos ajustes y soluciones permitieron superar los desafíos y asegurar el éxito del proyecto.

VI. CONCLUSIONES

- El uso del perceptrón como red neuronal proporcionó una solución efectiva para el procesamiento de señales de la cámara y la generación de comandos de movimiento. Aunque es una red simple, permitió al robot seguir la línea de manera confiable y precisa.
- El sistema de control basado en redes neuronales permitió al robot manejar eficazmente las variaciones en las condiciones del entorno, superando las limitaciones de los métodos tradicionales de seguimiento de líneas. Esto resultó en una trayectoria de seguimiento más precisa y un tiempo optimizado para completar la pista.
- El robot demostró una capacidad robusta para operar bajo diversas condiciones de iluminación, según lo mostrado en las pruebas con un luxómetro. Esta adaptabilidad garantiza un rendimiento constante y eficiente del robot en diferentes entornos luminosos.
- El diseño del sistema de control con la Raspberry Pi Pico W y el puente H L298 proporcionó una gran flexibilidad para ajustar los parámetros del robot. Esto permitió realizar ajustes finos en la velocidad y dirección de los motores, optimizando el rendimiento del robot en diferentes condiciones de la pista.

ANEXOS



Figura 7. Medición Corriente motor derecho sentido antihorario



Figura 8. Medición Voltaje motor derecho sentido horario



Figura 6. Medición Corriente motor derecho sentido horario



Figura 9. Medición Voltaje motor derecho sentido anhorario



Figura 10. Medición Corriente motor izquierdo sentido horario



Figura 13. Medición Voltaje motor izquierdo sentido antihorario



Figura 11. Medición Corriente motor izquierdo sentido antihorario



Figura 14. Medición corriente total del carro

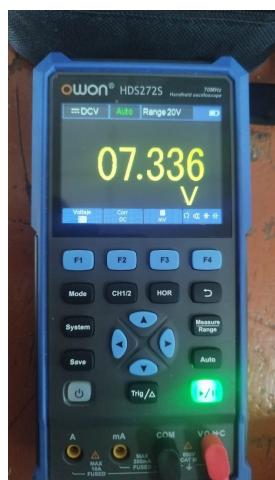


Figura 12. Medición Voltaje motor izquierdo sentido horario

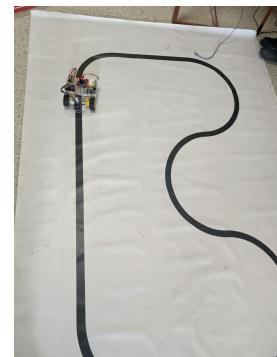


Figura 15. Montaje final del seguidor de linea

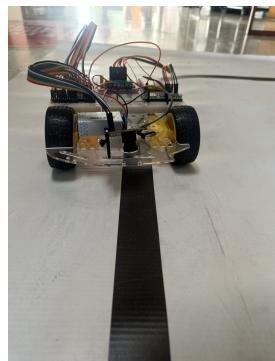


Figura 16. Montaje final del seguidor de linea

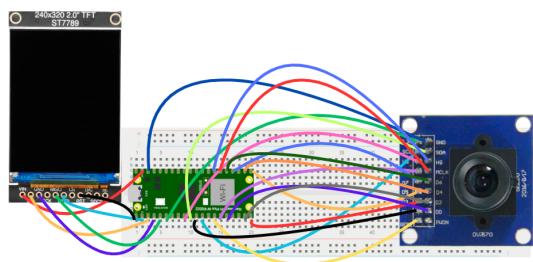


Figura 17. Conexiones raspberry - cámara OV7670



Figura 18. Visualización por pantalla Oled

REFERENCIAS

- [1] Advanced Motion Controls. (n.d.). Puente H. Recuperado de <https://www.a-m-c.com/es/experiencia/technologies/power-devices/puente-h/>
- [2] Repositorio Gerardo Munoz. <https://github.com/GerardoMunoz>